

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



带你深入Linux的世界，

让你的嵌入式和运维开发更上一层楼，对Linux系统的运用成竹在胸。

# 深入 Linux

## 内核架构与底层原理

刘京洋 韩方 著



中国工信出版集团



电子工业出版社  
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>



## 作者介绍



· 刘京洋 ·

中山大学双学士、硕士，创建中山大学嵌入式组，在嵌入式实验室工作6年。工作早期担任创业公司总经理，投资公司总裁助理，后来专心技术，先后就职于TP-LINK、YY直播和网易游戏，从事内核和网络安全研发，对Linux系统底层有深入的理解。联系QQ：575705195，很高兴与大家沟通探讨相关学术问题。



· 韩方 ·

武汉大学研究生毕业，先后就职于华为、YY直播，具有多年安全领域的攻防对抗、安全体系建设和开发经验，精通Linux内核开发和应用开发，申请过多项发明专利，多次参加国内外技术峰会并进行分享。







# 深入 Linux

---

## 内核架构与底层原理

刘京洋 韩方 著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING





## 内 容 简 介

本书主要描述 Linux 系统的总体框架和设计思想, 包含很多可以直接操作的实例, 目的是希望读者对 Linux 系统背后的逻辑有一个全面的了解。本书力求贴近实际的工作使用, 在比较核心且常用的技术点有更加深入的解释, 对实际使用 Linux 系统工作大有裨益。

本书共 13 章, 其中第 1~3 章是总览, 第 4~13 章是分领域阐述。第 1~3 章总体介绍 Linux 的基本知识; 第 4 章以 Linux 系统的启动开始深入叙述; 第 5 章是 Linux 系统运行中使用者最常接触到的进程概念, 重点介绍进程的原理; 第 6 章是 Linux 内核的内存管理方法与用户端使用内存的底层方法, 即重点介绍 glibc 底层到内核之间的内存管理过程; 第 7~13 章分别是关于安全、网络、总线与设备变动、二进制、存储、虚拟化与云、硬件专用子系统的内容。这些子系统都是 Linux 系统运行中非常重要的领域, 是深入理解 Linux 系统原理不可或缺的知识补充。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。  
版权所有, 侵权必究。

### 图书在版编目 (CIP) 数据

深入 Linux 内核架构与底层原理 / 刘京洋, 韩方著. —北京: 电子工业出版社, 2017.11  
ISBN 978-7-121-32290-7

I. ①深… II. ①刘… ②韩… III. ①Linux 操作系统 IV. ①TP316.85

中国版本图书馆 CIP 数据核字 (2017) 第 176716 号

策划编辑: 黄爱萍

责任编辑: 徐津平

印 刷: 三河市双峰印刷装订有限公司

装 订: 三河市双峰印刷装订有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 24.75 字数: 453 千字

版 次: 2017 年 11 月第 1 版

印 次: 2017 年 11 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn), 盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式: (010) 51260888-819, [faq@phei.com.cn](mailto:faq@phei.com.cn)。



# 推 荐 序

Linux 操作系统在超级计算机、互联网服务、桌面系统、移动和嵌入式设备等领域使用广泛，相关的从业人员和兴趣爱好者一直对 Linux 的理论和实践有较大需求。本人作为互联网领域的从业人员之一，非常荣幸可以提前阅读书中的内容，发现不同于市面上常见的 Linux 书籍，本书的内容从内核层面出发，对各个子系统的设计、实现和演化进行了梳理，并结合作者多年的亲身实践。以下部分是本人阅读后，希望与读者分享的一些感受。

第一个特点是**解释透彻**。Linux 发展至今已经超过 25 年，源代码融合了不同时期的演进和变化，因此回顾当时的背景，有助于更清晰地了解代码作者的意图和目标。以本书第 4 章“Linux 系统的启动”为例子，介绍了 BIOS 的局限性，如何向 EFI 演进的历史，另外还重点分析了 initrd 过渡根文件系统的来龙去脉，让本人对 Linux 启动的过程有更全面而深入的认识。又例如第 12 章“虚拟化与云”中的第二节，介绍了一段比较近的融合文件系统的历史，Docker 最初使用的 AUFS 逐渐过渡到合并到内核的 Overlay。

第二个特点是**实践性强**。在技术领域，实践往往能加快加深对相关概念的理解，本书有不少例子适合当作实验，感兴趣的读者可在单机环境或者虚拟机环境完成。例如上面提到的 initrd 文件系统的例子，书中比较完整地介绍了几种可行的制作方法。而在介绍 cgroup 的章节中，作者介绍了不同子系统可以限制的资源和效果，感兴趣的读者可以通过操作 cgroup 文件系统来观察实现的效果。再介绍第 8 章“网络”





## 深入 Linux 内核架构与底层原理

中的一个有趣的“六次握手”实验，这里可以看到一些很少见到的两端同时发起连接的 TCP 状态变化。

第三个特点是**指路明灯**。Linux 内核的子系统和模块非常多，覆盖的应用范围也很广阔，面面俱到显然是不现实的。作者希望更多地展示代码背后的思想，以及作者思虑后的理解，这比单纯的技术讲解更有营养价值，同时也鼓励读者在阅读时形成自己的见解，学会自己查阅相关的技术资料。例如，第 6 章“Linux 内存管理”中介绍了内存回收算法 PFRA，本人阅读前并不是特别清楚匿名页和命名页的不同处理流程，以及系统在内存低水位的行为，阅读后形成线索，可搜索出更多相关的资料。

第四个特点是**与时俱进**。近几年，业界利用 Linux 构建很多热点应用，本书在很多方面覆盖了 Linux 较新的功能，对从业者有较大帮助。例如，容器技术在应用服务上非常火热，本书在第 12 章“虚拟化与云”中对 cgroup 资源隔离和命名空间有基本的介绍。I/O 方面，本书介绍了能大幅改善性能的用户态 I/O，包括目前高性能网络中用到的 DPDK。在第 7 章“安全”中，还介绍了内核的 eBPF 虚拟机，很多新的内核调试工具、审计工具和高性能包处理都依赖这个机制。

总的来说，本人阅读后收获颇丰，对工作也有积极帮助，希望其他读者也能从中获取价值。最后，本人也是一名开源爱好者，感谢作者的辛勤付出，编写出一本内容详尽的 Linux 著作，希望 Linux 和其他开源社区发展越来越好。

李文俊



# 前言

要想深入研究并使用 Linux 内核，首先要知道 Linux 内核提供了什么，又能做到什么。很多初学者一进入公司就开始使用 Linux 内核开发内核模块，无论是使用通信方式、内存接口还是设备接口，都是早已被淘汰的内容。因为他们通常直接在网络上搜索一些很早之前发布的内容来指导自己如何完成开发工作，但他们手中却是最先进的内核代码。还有很多直接编写内核模块的人在嵌入式公司使用老版本的内核进行工作，虽然他们可能对内核之后的发展一无所知，但是他们能够一下子抓住主干，主干永远是在老版本的内核中就存在的东西。

很多刚入行的程序员认为自己能够征服一切，稍微在网上检索一下 Linux 的内容，就可以上手使用了。虽然写出可以用的程序不需要太多的知识积累，但是这么做相当于在信息不充分的情况下做决策。虽然一切操作系统理论的学习都不如实际去编写几行代码，但是理论又是十分重要的，因为它能够让经验升华成积累。

本书解释了 Linux 内核提供了什么，以及 Linux 系统底层是如何使用内核的。如果你对本书某一部分感兴趣，那么在深入阅读该部分的代码之前应先对该内容进行系统的学习，当你对内核系统有一个整体的把握时，方可挥洒自如。

本书的读者对象是有一定 Linux 基础的程序员，或者是有一定经验的嵌入式开发人员和运维人员。阅读本书像喝水一样，可轻松获得知识内容。若阅读本书遇到相对冷门的技术细节时，有兴趣的读者可以自行查阅其他相关资料。例如当列举文件系统的种类时提到 exofs，书中不会过多解释这个名词，因为大部分用户只关注它是文件系统的一种。





在学习 Linux 内核，阅读相关图书时候限定版本是不必要的，因为即使版本变化，原理仍旧可用。本书也会注明某个技术点之前是什么样的，现在是什么样的，未来可能是什么样的。人们更希望了解整个内核框架的内容，以及一些重要细节的深层原理。本书就将重点放在这两方面内容上，而并不局限于内核的版本，尽可能以最终被选择的解决方案作为实验重点。也就是说，本书所涉及的内核版本都比较新，但是也会观察从老版本到新版本过渡时内核在功能上的变化，比如 `ip rule` 命令在新版本中去掉了 `reject` 等 `action`。但是老版本的设计对于整体理解架构很有帮助，我们的根本目的是用实现抽象出概念，本书讲解的所有案例几乎都使用了占据较大市场份额的 Ubuntu。

感谢韩方，他对本书的出版起到了提纲挈领的作用，若没有他的帮助，我一定会被淹没在一堆技术细节中而不知道如何选择。他编写并且修改了部分章节，概览性质的图书最需要高屋建瓴的能力和丰富的经验，韩方在这方面非常强。

由于时间仓促，加之水平有限，书中的缺点和不足之处在所难免，敬请读者批评指正。

刘京详

2017 年 10 月

轻松注册成为博文视点社区用户 ([www.broadview.com.cn](http://www.broadview.com.cn))，扫码直达本书页面。

- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32290>



# 目 录

第 1 章 总览	1
1.1 简介	1
1.2 Linux 学习曲线和职业曲线	4
1.2.1 给自己定级	4
1.2.2 使用者	8
1.2.3 开发者	17
1.3 如何形成一个内核	24
1.3.1 内核形成过程	24
1.3.2 Exokernels 和 Anykernel	25
1.3.3 内核为何使用 C 语言	26
第 2 章 内核架构	29
2.1 常见架构范式与核心系统	29
2.1.1 Linux 内核上下层通信方式	29
2.1.2 横向系统和纵向系统	32
2.2 基础功能元素	32
2.2.1 模块支持	32
2.2.2 模块编程可以使用的内核组件	37
2.3 特殊硬件框架	39
2.4 特殊软件机制	41



第 3 章	内核数据结构	47
3.1	链表与哈希表	47
3.1.1	双向链表	48
3.1.2	hlist	48
3.1.3	ScatterList	50
3.1.4	llist	51
3.2	其他数据结构	53
3.2.1	树	53
3.2.2	FIFO 文件	53
3.2.3	位数组 bitmap	57
第 4 章	Linux 系统的启动	59
4.1	启动的硬件支持	59
4.1.1	固件	59
4.1.2	磁盘分区管理	60
4.2	Bootloader 和内核二进制	62
4.2.1	Bootloader	62
4.2.2	内核二进制	63
4.3	Linux 的启动原理	64
4.3.1	Linux 的最小系统制作和启动	65
4.3.2	initrd 文件系统	66
4.3.3	EFI 启动桩	68
4.3.4	启动管理程序	69
4.3.5	Linux 内核启动顺序	74
第 5 章	进程	75
5.1	进程原理	75
5.1.1	服务与进程	75
5.1.2	资源与进程	76
5.1.3	进程概念	76
5.1.4	父子关系	77
5.1.5	ptrace 系统调用	82



5.2	进程调度	90
5.2.1	调度策略	91
5.2.2	进程调度策略的配置	92
5.2.3	公平问题	92
5.2.4	内核线程的调度	93
5.3	资源	94
5.3.1	资源锁	94
5.3.2	资源限制	96
5.3.3	进程对系统内存的使用	97
5.4	多进程与进程通信	98
5.4.1	多进程模型	98
5.4.2	用户进程间通信	99
5.4.3	内核与用户空间的进程通信	103
5.4.4	Netlink 功能模块	105
5.4.5	其他 Netlink 种类	106
5.4.6	genetlink 的使用	108
5.4.7	inet_diag 模块	112
5.4.8	RTNETLINK	116
第 6 章	Linux 内核内存管理	121
6.1	内存模型	121
6.1.1	内存模型概览	121
6.1.2	内存组织方式	122
6.2	申请和释放内存	124
6.2.1	高端内存	124
6.2.2	设备内存映射	125
6.2.3	启动时内存的申请和释放: bootmem	126
6.2.4	Mempool	126
6.2.5	CMA (连续内存分配器)	127
6.2.6	伙伴算法	127
6.2.7	slab	127
6.2.8	用户端内存管理基础组件	128



6.3	内存组件	129
6.3.1	内存回收算法 (PFRA)	129
6.3.2	其他内存功能组件	130
6.3.3	内存压缩	132
6.3.4	BDI (backing device info)	133
第 7 章	安全	137
7.1	概览	137
7.2	密码学	138
7.2.1	密码学概览	138
7.2.2	摘要	139
7.2.3	加密	140
7.2.4	认证	141
7.2.5	数字签名	142
7.2.6	秘钥交换	142
7.3	Linux 用户和权限系统	143
7.3.1	系统启动时的权限	143
7.3.2	系统启动后的权限	144
7.3.3	内核中的用户和权限模型	145
7.3.4	Linux 安全体系	146
7.4	网络安全	148
7.4.1	netfilter 概览	148
7.4.2	Filter (LSF、BPF、eBPF)	151
7.5	函数调用的调试	163
7.6	内核调试	164
7.7	PAM 和 Apparmor	170
7.8	内核安全	175
7.9	常用安全工具和项目	175
第 8 章	网络	180
8.1	网络架构	180
8.2	socket	185





8.2.1	socket 简介 .....	185
8.2.2	类型与接口 .....	186
8.2.3	Linux socket 连接模型 .....	190
8.3	IP .....	191
8.3.1	IP 管理 .....	191
8.3.2	IP 隧道 .....	193
8.4	TCP .....	201
8.4.1	TCP 存在的原因 .....	201
8.4.2	TCP 的连接状态 .....	202
8.4.3	TCP 拥塞控制 .....	207
8.4.4	TCP 其他的功能特点 .....	215
8.5	网络服务质量与安全性 .....	217
8.5.1	TCP 安全性 .....	217
8.5.2	QoS .....	221
8.5.3	NAT .....	225
第 9 章	总线与设备变动 .....	229
9.1	PCI .....	229
9.2	USB .....	238
9.2.1	USB 概览 .....	238
9.2.2	USB 子系统上层 (USB 设备驱动层) .....	239
9.2.3	USB 子系统的中层 (USB core) 和下层 .....	242
9.2.4	Platform 总线 .....	244
9.3	用户空间的设备管理 .....	244
9.3.1	设备变化通知用户端 .....	245
9.3.2	设备类型 .....	247
9.3.3	内核数据结构的面向用户组织 KObject .....	253
第 10 章	二进制 .....	254
10.1	函数调用 .....	254
10.1.1	函数调用约定 .....	254
10.1.2	栈结构 .....	258

10.2	Linux 的二进制兼容性问题	260
10.2.1	进程的执行	260
10.2.2	同操作系统下的 ABI	261
10.2.3	内核版本	262
10.2.4	库	263
10.2.5	编译器	265
10.3	ELF 文件执行原理	265
10.3.1	ELF 文件分类	265
10.3.2	ELF 文件格式	266
10.3.3	进程加载器	280
10.3.4	链接与执行	281
10.3.5	ELF 文件的初始化	284
10.3.6	进程初始化前的加载	285
10.3.7	链接环境变量	287
10.3.8	内核加载 ELF	288
10.3.9	Audit 接口	289
10.3.10	一个简单的 ELF 解析程序	289
10.4	ELF 的安全性	292
10.4.1	二进制修改	293
10.4.2	二进制格式的病毒和木马	297
10.4.3	二进制安全特性简介	298
第 11 章	存储	301
11.1	磁盘管理	301
11.2	存储协议	304
11.3	通用块层抽象	314
11.3.1	通用块层功能概览	314
11.3.2	数据完整性校验	316
11.3.3	设备抽象	317
11.3.4	BIO 和 bio_set	317
11.3.5	request	318
11.3.6	request_queue	319

11.3.7	电梯算法	320
11.4	缓存层	330
11.4.1	BDI: 缓存设备	330
11.4.2	页回收	332
11.4.3	缓存机制	334
11.4.4	缓存页的状态	337
11.5	文件系统	338
11.5.1	文件系统的种类和选用	339
11.5.2	拥有特殊功能的文件系统	339
11.5.3	其他领域的文件系统	343
11.5.4	文件系统的意义	344
11.5.5	文件系统的抽象: VFS	345
11.5.6	ext4	346
11.6	存储系统	348
11.6.1	存储形式	348
11.6.2	存储格式	350
11.6.3	分布式存储系统	350
第 12 章	虚拟化与云	358
12.1	常见的虚拟化方案	358
12.2	虚拟文件系统	361
12.3	cgroup	363
12.4	Docker	369
第 13 章	硬件专用子系统	374
13.1	无线子系统	374
13.2	音频子系统	380



# 1

## 第 1 章

### 总览

#### 1.1 简介

在 Linux 内核出现之前出现过很多优秀的内核，甚至同时期直至今日，那些竞争关系的内核仍然存在。2007 年我刚学习 Linux 时，还在犹豫是否要学习和使用 FreeBSD，现在我绝对不会有疑惑了。可以说 UNIX 是 Linux 发展壮大的原因，Minix 是 Linux 发展的方法。Linux 从对 Minix 的深入研究发源而来，但是实际地推出大量借鉴了 UNIX 的交互方式和技术模式。UNIX 本来也完全可以压倒性地压制 Linux，因为那时的 UNIX 内核有多个封闭版本，并且都是商业运行，很多非常强大。Linux 之所以能够后来逐步超越 UNIX，是因为 Linux 的开源属性和 Torvalds 的杰出领导。

直到今天一个软件想要获得最广泛的应用，开源仍旧是不二手段。通过这种方式能够让自己的软件取得比更加友好强大的商业软件更大的市场空间。但是代价就是难以找到盈利模式。UNIX 就像一个创新发源地，很多需要花费巨资的 Linux 的标准化工作，例如 POSIX 都由 UNIX 背后的企业去完成，一些新奇的技术也是由在商业系统研究的过程中发表的 paper，被 Linux 直接借鉴实现。近年 Android 的发展也让 Linux 从谷歌获得了大量的改进和修复（最典型的是 cgroup）。久而久之就形成了 Linux 独特的价值观：求全第一，求精第二。

Plan9 是由贝尔实验室开发的系统，参与阵容可谓豪华：Rob Pike（现在在 Google

工作，负责 Go 语言的开发)；Ken Thompson (C 语言和 UNIX 创始人)；Dennis Ritchie (C 语言和 UNIX 创始人)；Brian Kernighan (awk 之父)；Doug Mcilroy (UNIX 管道提出者，UNIX 开发参与者)。我在使用 Linux 时最大的震撼就是一切皆文件的思想，然而这在 Plan9 中已然是价值观上的定海神针。Plan9 中最本质的思想是“一切皆是文件”，甚至 CPU 是一个文件；内存是一个文件；网络是一个文件，任何的东西都是一个文件。这在 Linux 中已经尽量做到这一点，在 Plan9 中被极度地强化。

Linux 和 UNIX 是一个多用户分时操作系统，就是多个用户共享一个操作系统资源。不管是 CPU、内存、网络，都需要通过调度器分配调度。比如 A 机器的文件需要使用 B 机器的 CPU 来处理，方法就只有通过某种协议，将 A 机器的文件下载到 B 机器中，然后 B 机器处理完以后再回传到 A 机器中。Plan 9 的“一切皆是文件”看起来很好的解决了这个问题。A 机器想使用 B 机器的 CPU，只需要将 B 机器的 CPU 挂载到 A 机器的 CPU 的文件中就能完成这个需求了。当然，两个机器之间也有一个协议“9P”来进行文件挂载和表示，但是这个操作对上层的操作系统来说已经是透明的了。

Plan9 是一个分布式操作系统，它能把网络上一切资源当作文件来进行使用，这其实就是云的概念了。但是看起来好的东西在实际使用中不一定好，很多时候人们会更倾向于使用 socket，就像它本可以用 Python 语言一句话完成一个 socket，但是它在某些极端情况下还得用 C 语言的 socket，因为那样更加可控并且效率高。当你挂载了远端的 CPU 到本机时，用起来像用本地的 CPU 一样，这看起来是优点，但同时也是缺点，使用这个系统的人会莫名其妙地发现自己的程序很慢，但是他很难想到是被调度到远端的 CPU 上去执行了，如果想要加速程序运行速度，就得深入地研究 CPU 挂载的流程和原理。实际上不是那么方便，这有时候就是对失误的一个很好的阻碍。

我们做云系统的时候就会发现，很多时候为了可靠性和稳定性都会在一定程度上牺牲易用性。一个操作系统最重要的素质就是稳定性，否则任何的上层进程出了问题，还得去找系统级的原因，那样就会极大地阻碍这个系统的实际有效应用。

除了 Plan 9 之外，Linux 系统还有很多重量级的竞争对手：Haiku、BeOS、Amiga、OS/2、Arthur、XTS-400、Infemo、Symbian、Palm OS 等。Linux 系统目前在嵌入式和服务器领域取得的成功并不是因为它一开始就是这个定位，而是只有这两个大方向显著适合 Linux 系统，因为 Linux 系统什么都能做。而其他的竞争对手通常都有共同的特点，那就是专业或者商业。比如 BeOS 为多媒体而专门设计得十分精良，还差一点被苹果公司用作桌面系统，它的定位可以说是非常精准，但是市场无数次

的证明了胜出的不一定是好的。但是其派生的开源系统 Haiku 延续它发展了一段时间，但是也失败了。就连微软、IBM 联合重金研发，诞生的当今软件工程圣典《人月神话》的 OS/2 项目也迅速归于失败。Inferno 操作系统继承自 Plan 9、Amiga 系统同期拥有不亚于苹果的技术实力，也都因为过度依赖于某个公司的开发，经营不善而没落。我们熟知的 Palm OS 和 Symbian 也都饱尝了封闭拒绝进步带来的恶果。1990—2000 年是一个时代，这个时代产生了许多未来操作系统的种子选手。

信息行业几乎已经证明了一件事，在一个领域几乎只能容纳一两家巨头。那时的先驱者不一定认识到这个问题。他们或者是因为经营问题、销售问题、定位问题、技术问题，又或者是因为资金问题，他们都失败了。Linux 系统走了完全不同的路，这条路可以说是操作系统选手里面 Minix 最早尝试的，但是有完整的源代码。无疑是 Minix 的这种模式直接导致了 Linux 的诞生，其他操作系统走过的弯路，Linux 不必再走。Linus 本人是一个有情怀的技术人，他没有选择试图成立公司，他可能失去的是一个伟大的公司，但是他却收获了一个伟大的产品，并且极大地促进了整个人类信息技术的进步。

Linux 系统是一个工具，也是一个现象、一个平台。它很可能是最近 20 年对信息产业贡献最大的开源项目。路由器、手机、监控系统、互联网的服务器，甚至桌面系统都逐渐的在应用 Linux 系统，甚至除了桌面系统，在大部分的信息系统里，Linux 系统已经几乎成为了唯一的选择。大部分的应用场景都是使用 Linux 作为一个操作系统平台，而在之上运行企业自己的应用。根据这个应用对底层的需求层次不同，开发者对 Linux 知识了解程度的需求是截然不同的。纯粹逻辑的应用是几乎完全不需要感知到 Linux 的存在。编写逻辑，编译部署到 Linux 系统上就可以运行。但是包括运维、嵌入式，尤其是涉及更高的性能要求或者稳定性要求等时，就需要借助 Linux 内核或者系统底层提供的机制了。

Linux 内核提供的机制比较常用的部分，在用户端基本会有底层库进行封装提供，甚至一些内核没有提供的能力，底层库也会提供。所以当你需要底层知识时，可能使用的是底层库提供的机制，也可能是底层库提供不了的，此时就由内核直接提供。通常情况下底层库都需要尽可能地封装内核提供的功能，但是由于 Linux 内核的特性，它尽可能地集中了功能，并且直接通过文件系统接口暴露，这就导致了例如 /proc、/dev 等文件系统的使用可以非常方便地直接调用到内核的功能，底层库也就没有必要封装了（虽然大部分它们也都封装了）。

所以当我们深入到 Linux 时，通常会同时接触到底层库和内核的用户空间接口。



而要使用这些底层功能，通常需要对这些功能本身的实现原理有一定的了解。本书就是在这个点上提供的知识分析。

## 1.2 Linux学习曲线和职业曲线

### 1.2.1 给自己定级

鉴于读者可能分布在不同的知识水平阶段，所以首先要做的是对号入座，将自己对 Linux 的掌握与 Linux 相对全貌对比。有一些人是某一些领域的技术专家，有一些读者可能是接触 Linux 不久，还有人可能是找不到前进的方向。在学习 Linux 的过程中，最大的问题不是自己学不会，也并不存在由于智力问题导致自己无法提高的现象。所出现的问题主要有两个：一个是不知道学什么；另外一个学习的内容没有实际应用的场景，从而导致无法透彻地领悟。第一个问题是本书希望解决的，第二个问题则需要各自在工作岗位上不断练习。

想要以 Linux 谋生的，下面介绍的“专家级”之前的知识一般都要学会，因为在工作中都会用到，但是也绝对不是所有的知识点没有掌握就没法工作。这里的定级也限于对 Linux 操作系统的使用，一般是软件和系统特性，也并没有涉及太多的编程要求。编程是另外的一个层面，编程能力不好不代表 Linux 的使用水平不行，所以这里的定级不怎么考虑编程水平。并且这里所列举的条件和分级都不全面，只是一个难度的代表，也并不是详细规范的分级制度，仅仅具有参考价值。并且本分级偏向从零开始自学 Linux，然后使用 Linux 去实际完成工作的人。有的人一毕业就开始开发 Linux 内核也完全可以，甚至可以更加深刻快速地学习和理解整个系统。但是并不是所有人都有这个机会。所以本定级用由浅入深的自学流程进行分级。

#### 1. 未入门

大部分学习信息技术的人几乎不可能没有听说过 Linux 系统，大部分人也都使用过 Linux 系统，很可能第一次接触的是 ls 等命令接口，或者是 Ubuntu 的友好桌面环境。这些人不能被认为已经对 Linux 的知识达到了入门级的水平，也统一归到“未入门”的水平。如果是自己研究 Linux，未入门这个级别需要额外突破以下两个条件。

- 不会安装操作系统。



- 不会用虚拟机（安装和使用）。

## 2. 入门级

大部分希望学习 Linux 的人首先接触的都是些基本的命令，随着图形化的发展，很多初次接触 Linux 的人可能首先接触的是图形界面的使用。不像早期接触 Linux 的人，只要使用图形界面就会遇到无数的问题，需要自己去解决。早期的 Linux 使用者，刚使用 Linux 的时候就是以 Linux 系统问题解决者的姿态开始的，而且早期发行版没有在市场上决出胜负，大家很可能使用的是各种奇奇怪怪的发行版。例如我使用的第一个发行版就是 Everest，现在的大部分人都没听说过这个发行版。但是即使是现在，专用发行版依然有良好的存续空间，但是随着几个主流发行版的发展，这个现象很可能会快速改变，所以目前要达到入门级水平，要求做到以下两点。

- 熟悉常见的发行版。
- 会一些最基础的命令（例如 cd、ps、top、ls、ifconfig 等这个级别的）。

## 3. 基础级

当我们使用 Linux 系统时不再是一直在执行 ls、cd 这些命令，感叹于命令行的黑科技的时候，我们就会进一步挖掘一些 Linux 的用法。

- 可以使用一些常见的命令（touch、tail、date、find、du、fdisk、less、pidof 等这个级别的命令），使用这些命令标志着你知道了 Linux 操作系统的核心工作是资源管理。
- 对图形界面操作得比较熟练，并且能够对应一部分的后台命令。
- 知道一些常用的配置文件的作用（如/etc/hosts、/etc/resolv.conf、/etc/passwd+、/etc/mtab 等）。

对 Linux 系统的加深理解是随着自己对操作系统知识的加深而加深的。毕竟 Linux 只是工具，对 Linux 系统理解的加深也是对操作系统理论知识理解加深的过程。很多人学习 Linux 并不是学习操作系统原理，而是通过知道一个命令，然后看看这个命令能干什么，从而来逐步了解操作系统。这种属于搜索式的学习方法，也是比较符合初学者学习思路的方法。但是最有效的方法却是在读英语书时经常能读到的思路，就是系统性地把“大部头”“啃”下来，然后根据书里的内容快速加深自己的认知，但是初学很容易迷失在一大堆的定义之中，不如实际的操作来得深刻。

由于属于基础级水平的命令特别多，这里不可能全部列出来。这个级别的命令

在你学习了更高级的命令时就知道有什么区别了。例如你不会把 `lsusb`、`setarch` 等命令列入基础级，因为它们的专用性非常强。属于基础级的命令通常是无论是谁都要掌握的基本命令。

#### 4. 中级

一般自学 Linux 系统不会产生这个级别知识的需求，而在工作中就会频繁遇到，这些工作中需要的目标相对明确的知识就是中级的知识。

- 掌握一些运维时常用的命令（如 `iftop`、`iptraf`、`rsync`、`ngrep`、`sar`、`acct`、`sg` 系列、`inotail`、`nmap`、`lsof`、`ip`、`dig`、`wall`、`write`、`mkfs`、`grub` 系列、`awk`、`sed`、`cron`、`ldconfig`、`chrt`、`renice`、`nohup`、`socat`、`ethtools` 等）。这些命令的显著特点是你需要使用 Linux 去完成一定的任务。我初学 Linux 时买过一本《Linux 命令速查手册》，我认为这里的命令大都属于这个级别需要掌握的。
- 熟悉一些高级配置文件的使用（如 `ld.conf` 配置、`sshd` 配置、`samba` 配置、`PAM` 配置、`vsftp` 配置等）。《鸟哥的私房菜——服务器架设篇》这本书里面的内容相对基础，适合处于这个级别的读者进行阅读。但是这个级别的配置文件的知识并不限于此。
- 熟悉 `Proc`、`Sys` 文件系统（进行诸如 TCP 调优、`Core Dump` 目录和文件更改、`PCI` 设备的 `bind` 与 `unbind`、电梯算法调整等工作），这些操作并不容易用到，但是如果你完整地掌握了 `Proc`、`Sys` 文件系统的用法，你将会非常出色。
- 熟悉 `uevent` (`hotplug`)、`inotify`、`iptables`、启动时的保留内存 (`CMA`)、`key-ring`、`ACL`、`sg` 等高级系统特性。中级水平的人应该在一定程度上使用到常用的系统特性了，而不是像做普通的后台开发那样并不太关心系统的特性只关心逻辑。合理地使用系统特性并配合 `Shell`，可以出色地解决嵌入式开发和运维的一些相对高难度的需求。
- 懂得根据自己的需求配置编译内核，可以使用脚本编程。配置自己的内核的能力几乎是这个级别必备的素质。不详细配置内核，几乎不太可能对内核有通篇宏观的了解。而不会脚本，在 Linux 的世界里就像不会走路一样。

这个级别对于大部分国内专业使用 Linux 的运维工作者来说很多是没有逾越的。很多运维工作者甚至不知道 `inotify`、`uevent`，甚至一些关于 `Proc` 和 `Sys` 的知识也只是知道一些常用的知识点。他们平时干的更多的是写 `Shell` 脚本，调用一些常见的命令，完成一些自动化的运维工作。这个阶段的学习是最有趣的，也是最能够深入了

解 Linux 一个过程。运维如果能够熟练使用并且理解 socat, 就基本上达到了对系统知识的要求了。例如 `socat exec:"tailf /var/log/auth.log" TCP4:1.2.3.4:1234&` 一个命令就完成了日志收集的程序。

## 5. 高级

属于高级水平的人员就有一定的专业化倾向了, 但还称不上彻底专业化。这部分的知识是一个顶级的运维人员在突破了中级水平的同时需要掌握的内容。没有这部分知识的支撑, 就不能很好地定制 Linux 系统。由于这个级别 Linux 的使用一般离不开网络, 所以这一部分的网络进阶将是重点。

- 可以掌握系统的高级特性。如安全的 Capabilities、suid、PAM、LSM; 如详细的 Kobject 体系 (可以通过查看 Sys 文件系统取代常用的命令)、Netfilter 的 Hook 使用和 BPF (tcpdump 和 eBPF 的使用)。能够熟习中断系统、合理地绑定中断源、进程的优先级调度 (几种实时优先级和非实时优先级的设置)、OOM 分数的调整、cgroup 等。
- 可以自由地选择使用高级的文件系统以服务于自己的需求 (如 Squashfs、Gfs、Ecryptfs、Configfs、Overlayfs 等)。
- 熟悉 ssh (转发、反弹等高级操作)、nc、socks、tcp wrapper 等远程访问相关知识, 可以做深入一些的安全审计和配置, 需要更加深入了解 iptables 的各个模块, 例如 xt\_connlimit、ip\_set 等相关的几十个网络过滤模块的使用。
- 熟习 TCP 的内核内部实现原理, 常见的拥塞算法, 甚至常见的 TCP Offpath 攻击, 还有对应的一些与 TCP 相关的 RFC。要求几乎是完整地看过 TCP 相关的代码, 读完了相关的 RFC, 并且理解其中的算法。
- 可以进行内核的高级定制编译 (适当地修改内核代码, 追踪解决内核 BUG)。
- 可以编写基础的内核模块, 使用常用的内核编程接口。
- 懂得二进制原理 (ELF 文件格式、objdump、ld、nm、strings、readelf 等常用的二进制软件)。
- 各种 Bootloader 的制作和安装。
- 熟习 GCC 编译的选项和技巧, 可以写链接脚本, 自己安排 Section、Segment 等。
- 需要会使用更多的工具, 比如 slabtop、ltnng、blktrace、perf、tc、tune2fs 等。



## 6. 专家级

系统的了解和预备知识的准备同样重要。例如 TCP 的深刻理解才会知道如何做 SYN 的 DDOS 防护（不是内核的那个简单的开关）；对无线理论的深刻了解，才能会无线的内核和应用内容。对于这部分知识的掌握，核心的工作已经不在于掌握 Linux 是怎么做的了，而是掌握这部分知识本身，Linux 只是实现了这部分知识的要求。类似 Wi-Fi 或者显卡这种至今仍旧采用闭源战略的企业方向，Linux 实现了兼容就已经非常不错了，就不要挑剔性能和稳定性了。常见的专用子系统有内核虚拟化、内核存储逻辑（SCSI、PCI、USB 等）、无线子系统、音频子系统、显卡子系统、电源管理子系统、网络子系统、电源管理子系统等。

本书会重点涉及存储和网络两部分专用子系统。内存管理虽然也可以说是子系统，但更多的内容是高级技能，很少有人是专业做内存管理方面工作的，类似的还有进程子系统。

## 1.2.2 使用者

在 Linux 的使用者中有用来替代 Windows 的普通桌面用户（如使用 Startos、Deepin 或 Ubuntu Desktop）。使用者中也有命令使用者。这部分人涵盖了很多实际的工作岗位，典型的是运维人员。运维人员有很多细分的子岗位：系统管理、软件管理、后台服务器管理（例如数据库、Http Server）、软件部署等。Linux 用好了可以谋生，目前几乎所有的后台开发都需要或多或少地掌握 Linux 相关知识，因为国内互联网后台几乎被 Linux “统一”了。

### 1. 桌面使用者

使用桌面可以只使用鼠标进行点击操作即可，但是桌面也可以被用得比较深入。也有专门的职业工作就是怎么把桌面用起来。例如嵌入式里的裁剪和启动桌面（比如让 startx 命令跑起来）。

Linux 的桌面有很多。普通的使用者一般会直接使用几个常见桌面提供的环境和软件，而不同于 Windows 的鼠标点击，桌面的使用还可以更加深入。

#### （1）发行版

Linux 的发行版有很多，一般是不同风格或者是服务于不同目的的专用发行版。



常见的通用的发行版有：Ubuntu、OpenSuse、Fedora、Debian、Mandriva、Mint。目前越来越多 Linux 的使用者和开发者转向 Ubuntu，甚至以前在服务器领域常用的 Centos 也在慢慢地丢失市场，而 Ubuntu Server 的市场占比开始增长，甚至 Kali 这种专业性极强的安全渗透发行版，其安全能力也在逐渐被 Ubuntu 追赶。但是近期的 Ubuntu 放弃 Unity 事件，使得市场的未来前景仍然充满变数。

专用的发行版如 Kali（网络渗透）、Puppy、LUbuntu（精简）、Coreos、Ubuntu core（虚拟化）、Router OS（路由器）。这些专用发行版一般提供给专业用户，要发挥其全部威力，通常需要更多的行业知识。

## （2）桌面环境

常见的桌面环境有很多：Ubuntu 的 Unity、Gnome、Kde、Cinnamon、Mate、Lxde、Xfce。一般各个发行版都会携带不同的桌面环境，每个桌面环境的窗口风格都是不一样的。包括随桌面管理器携带的配套软件一般也是不一样的（例如计算器、包管理器、音乐播放器等）。但是有的流行的软件还是会被移植到不同的桌面管理器上的。

Gnome、KDE、Unity 的使用者最多。Unity 目前只用于 Ubuntu。Ubuntu 也并不是只支持 Unity，对于几乎所有的桌面环境，Ubuntu 都有对应的支持版本。Ubuntu 放弃 Unity 之后采用 Gnome，也充分说明了未来对于桌面环境市场的预期变化。

高级的桌面使用者一般可以自由选择不同的桌面环境，理解每个桌面环境工作的原理，理解 X 系统，可以远程多终端使用 X 系统，自由选择启动，甚至不启动 X 系统或者 X 系统的一部分。理解 X 系统和桌面管理器与窗口管理器的区别。

存在如此多的桌面环境就导致发行版在选择桌面环境的时候的质量参差不齐，也由于 Linux 发行版的创作者大多数都是不盈利的小团队，甚至是个人。所以他们倾向于使用容易处理的小型桌面环境，所以你会发现大部分的发行版都是使用的比较原始脆弱的小众桌面环境，虽然他们的作者是一位很有能力的“Linuxer”。这种趋势随着 Ubuntu 等大型发行版的商业化运行被逐渐改变，分布式的自由发展，在遇到一个人工作量难以覆盖的大项目，并且缺少强有力的领导时，逐步让步于集中式的商业开发。

## （3）FrameBuffer

FrameBuffer 是 Linux 图形界面显示的兼容方法，但不是最高效的，因此这个机制一般作为最后的显示手段。一般的显卡驱动都有自己的专门机制来支持。FrameBuffer 模式的显卡本身不具有任何运算数据的能力，它好比是一个暂时存放水

的水池。CPU 将运算后的结果放到这个水池，水池再将结果流到显示器。中间不会对数据做处理。在这种情况下，所有显示任务都由 CPU 完成，CPU 的负担很重。从 FrameBuffer 这个名字我们就能猜测到它的功能了（帧缓冲）。

在 Linux 内核中也有 FrameBuffer 机制，模仿 FrameBuffer 显卡的这种功能。它的好处是把显卡的硬件结构抽象掉，把所有的显卡都当作一个“水池”来用。应用程序也可以直接读取这个水池的内容。FrameBuffer 的设备名是 `/dev/fb`。使用 GPU 的时候理论上确实可以把 GPU 的内存也映射到这个抽象设备上，但是很少人会这么做。

Linux 字符界面在高分辨率下，启动时会会有一个小企鹅的 Logo，这个 Logo 就是用 FrameBuffer 功能写在屏幕上的，因为此时显卡尚未保证初始化。

#### （4）X 协议

Linux 内核给用户提供了显示 FrameBuffer 设备用于显示，所有要显示的东西写到 FrameBuffer 去就好了。也就是内核提供了显示的机制，但是没有提供显示的内容。所以显示内容需要操作系统去实现。

几乎所有的 Linux 显示的核心都是 X 协议，X 是一种显示协议，实现这个协议的常用软件有 Xfree86、Motif（商用）、Xorg、Xnest 等。所以 X 协议也可以在 Windows 系统中实现，苹果操作系统也是用的 X 协议，只是是实现在内核中。现在的 Linux 发行版一般都默认使用 Xorg。好多人都看到“X11”这个词，X11R6 实际上是 X Protocol version 11 Release 6（X 协议第 11 版第六次发行）。

## 2. 命令使用者

命令行是 Linux 的最出彩的 API 接口，从某种意义上说命令行就是面向组件的编程，并且这些组件就直接是一些可以单独运行的程序。我们知道编程只有复用了才会有最大的效率，而 Shell 就是把编程复用到到了程序级别。应该说在编程思想上是非常先进的。

但是复用到程序级别就必然有一些程序上的开销损耗，导致 Shell 程序的性能不会太高，但是考虑到被复用的组件，例如 `grep`、`sed`、`awk` 都是有着极高的效率，所以这种复用通常会随着组件性能的提高和机器性能的提高而使得性能问题越来越可以被忽略。

命令使用者就像在写 Python 程序时候的交互式模式，只是 Shell 在交互式上做得更好。使用命令就是使用程序，只是在 Linux 上通常是指 Shell 脚本程序的文本形式，例如 Busybox 这种工具还可以在一个程序的内部集成本来由多个程序完成的组

件级的复用。仅仅是一个一个地使用命令不能够体现 Linux 这个 Shell 模式的强大，更多的时候是使用 Shell 编程语言进行编程。这种程序本质上也是由一个一个的命令组成的。研究一下 Shell 编程就会有一个体会：你写的每一个语法本质上都是一个命令，只是语法层次上的命令都是直接集成到 Shell 程序内的。最常用的 Shell 程序和语法是 Bash 带来的，但是除此之外还有很多语法不同的 Shell 实现，例如 csh、zsh 等。

### 3. 运维使用者

运维人员可以说是最专业的 Linux 使用者了，因为他们要关心 Linux 整个系统的运行状况，是对 Linux 系统的使用方面挖掘得最深的人。研发人员可能会更深入地研究 Linux 系统，但是在广度方面一般不如运维人员。运维一般是指互联网公司需要 Linux 服务器后台日常维护工作人员，与后台开发有本质的区别。

一个运维人员的基本功应该是查看 Linux 系统状态的命令和脚本的编写，深入一些的运维对 Linux 有更深刻的了解。有经验的运维人员除了掌握这些知识点之外，还可以处理 Linux 系统常见的问题和故障，会使用运维系统的设计和 DevOps 等先进的协作方法。以下几个知识点建议运维人员掌握。

#### (1) 命令

基础的命令是每个命令行使用者都要掌握的，然而由于运维这个职业更多要关心的是机器的资源运行使用情况，所以有一些更专业的常用命令。由于运维工作的重点在于资源，而操作系统的资源大部分就是网络、内存、磁盘、CPU、文件句柄这几大类，所以运维的大部分工作就是用不同的命令监控这些不同类别的资源，并且做一些脚本化的操作。一些常用的基础命令，例如 top、ps 等就不属于运维这个职业的命令，而应该是属于使用 Linux 系统命令行都应该掌握的基础命令。以下列出的命令也相对基础，很多专用的高级命令会在书中给出，还有很多专用系统级的已经不能称作是命令的，例如 Nagios 工具也不在此列。并且必须要强调的是资源监控的命令大多数都是基于 proc 文件系统的内容，如果你深入研究了 proc 文件系统，你将可以自己独立写出很多专用命令。

#### ① 网络

- iftop: 查看连接流量。还可以交互地查看端口到端口，以及进行过滤。
- netstat (可以用 ss 或 lsof 替代): 查看网络的连接状况。
- iptraf: 图形化的观看 IP 流量。
- nethogs: 类似网络的 top 工具。



- tcpdump: 抓包直接打印或者保存为 pcap 文件, 甚至可以生成 bpf 代码。
- ngrep: 把网络数据包当成 grep 文件一样过滤。快速查看网络数据的好工具。
- mascan、hscan、nmap: 扫描器。
- nping: 具有很高灵活性, 且功能广泛的网络调试工具。

## ② 内存与 I/O

- vmstat: 报告内存的使用情况。
- iostat、iotop: 报告磁盘 I/O 的使用情况。

## ③ 进程

- htop: 增强的 top, 界面更漂亮, 功能也更多。

## ④ 其他

- sar: 综合性地查看系统资源使用的工具。
- lsof: 列出打开的文件。这个工具非常强大, 这是因为在 Linux 系统下一切皆是文件。
- acct: 用于监控用户。
- monit: 一个比较全面的系统资源监控命令。

除上述之外, 还有一些用起来比较方便的工具, 例如 nping、incron (使用 inotify 机制, 当文件发生变化时自动执行注册脚本, 对应于 cron 是基于时间的, incron 是基于文件事件的)、rsync (远程文件同步)。这类工具是海量的, 运维人员会在对它们越来越深入地使用中逐渐遇到新的需求。sleuthkit 工具是典型的专用工具, 这类的工具更强大, 但较少用到, 它是高级命令使用者需要掌握的工具。

## (2) 机制

系统机制主要是指一些系统目录和系统基础功能的使用。典型的系统目录是 /proc 和 /sys 目录。proc 文件系统和 sys 文件系统可以作为运维人员的提高篇进行学习, 可以从 /proc/pid/ 下面的文件中看到在上述命令中看到的東西; 从 /proc/sys/ 中看到和修改系统当前的参数配置; 从 /sys/ 目录下看到系统当前的物理资源 (例如通过 rotational 文件来判断一个存储设备是否是 SSD), sys 文件系统内部的 Kobject、Ksystem、Kset 机制和 uevent、hotplug、udev 等要熟悉。最好是手动遍历 /proc 目录和 /sys 目录下的所有文件, 了解每一个文件的作用和使用方法。例如如果你想找系统启动的时间, 除了 ps 命令之外, 你可以直接查看 /proc/pid/stats。

系统的基础功能与命令工具的边界比较模糊, 实际上很多基础功能都是由一个个软件包组成的, 各个软件包又分别对应着一系列的命令。一般比较优秀的运维人



员可以自己升级 Linux 内核，能够熟练使用 grub。可以熟悉使用 inotify、rsync、ZoneKeeper、Ansible 等基础设置软件进行发布和同步。基于对 SCSI 在 Linux 的重要性，需要了解 sg 系列命令的使用。例如一个典型的跳板机的配置，配置如下。

```
#ssh_config
Host * !jumper #其他机器的访问规则
    User liujingyang
    ForwardAgent yes
    ProxyCommand ssh jumper nc %h 12345 2>/dev/null
    IdentityFile /home/archerbroler/Identity_1
    PasswordAuthentication no
    Port 12345
    StrictHostKeyChecking no

Host jumper #跳板机的访问规则
    User liujingyang
    HostName jump.huanjushidai.com
    IdentityFile /home/archerbroler/Identity
    PasswordAuthentication no
    Port 12345
    StrictHostKeyChecking no
```

这看起来是 ssh 软件的使用，实际上还涉及 nc 命令，大型企业都会遇到跳板的实现原理，以及 Ansible 的使用配置等。你还可能在设置这个文件时跟 iptables 打交道。

### (3) 脚本

运维人员喜欢让自己的 Shell 尽量“帅”起来，例如使用 Guake、Tmux、Zsh、Emacs 等，还要熟练使用 ssh 远程管理系统，以及相关配置。通常大家都是编写 Bash 脚本。

cron 命令和自动化脚本是管理每台机器必需的，除非非常大的公司有自己独立的运维系统。脚本的编写也将运维人员分为好几类，由于每种 Shell 的语法实现是不同的，所以不存在标准的 Shell 语法，只存在使用广泛与不广泛的区别。最常用的是 Bash，但是最强大的当属 zsh。你也完全可以自己用 Python 实现一个自己定义语法的 Shell 解析器，只要它能调用命令并且具备逻辑能力。

我们都知道当你只是使用一个工具的时候，你会希望那个工具越简单好用越好，但是如果当一个工具你要一辈子使用它的时候，你会更倾向于那个工具要尽可能的

强大，甚至不惜为此付出很多的资金成本和时间成本，而复杂度通常和强大是同时出现的。Zsh 用不好就会慢，并且难以找到懂行的人来维护。所以从社会的角度看，Zsh 不能被广泛地传播是 Bash 这种良币驱逐了劣币。但是从技术的角度看，很多技术人员就会认为是劣币驱逐了良币。

#### 4. 系统管理员

系统管理员与运维人员很类似，不过运维人员一般出现在互联网企业，系统管理员一般出现在传统企业。系统管理员比运维人员更偏向于使用现有工具，而运维人员对系统的了解和脚本的使用会比系统管理员更熟悉。

系统管理员一般对升级内核要求不多，但是升级系统版本还是有的，库的部署、部署环境（Docker）、解决环境问题、软件发布、配置管理等。一般对 `etc` 目录下的配置文件都要很熟，一个占用时间的工作很可能是修电脑。但是也有很多技术超群的系统管理员，可以管理高度复杂或高度定制化的系统。但是在现实中，通常系统管理员要承担的工作会超出界限，在实际中并没有完全精准定义的系统管理员的这个职业。例如追查黑客入侵、评估测试购买硬件、服务器权限管理等难以界定的工作，很多时候都是谁可以解决就找谁完成了。

#### 5. 后台服务管理员

后台服务管理员中最常见的职位是 DBA，就是 Database Administrator。数据库管理员是所有后台服务中最常见的，很多企业的数据库是外购的企业级的数据库。对于这些数据库的管理就相对简单，更多的侧重于使用。主要工作是安装、监控、备份、恢复。但是现在 MySQL 和一些开源的 NoSQL 数据库占领企业市场的比重越来越大，导致数据库的管理员多了很多技术人员的“味道”，这样就很难界定什么工作属于谁了。例如数据库的集群、分库分表、性能优化等工作就成为数据库管理员和后台开发人员共同关心的问题。

诸如数据库管理、Http Server、NTP、DNS Server、FTP Server 等各种常见的服务器的搭建和配置管理，说起来容易，但每个软件都有很多配置文件，并且大多数也都不像给用户使用的产品那样友好，而且都有一些“坑”要踩。只有详细阅读文档，多尝试，相关技能才会有提高。

#### 6. 安全使用者

Linux 的安全系统发展至今很全面，但是还远远不够。Linux 系统距离一个安全

的操作系统还有很长的路要走。一般情况下，系统安全和业务安全是被纳入运维工作范畴的。大企业会有专门的安全系统开发人员，但几乎很少有专门的安全运维人员。这是一个专用领域，虽然目前重视程度不太高，但是在 Linux 中安全系统是一个很大型的系统。

系统安全也分为两部分，一部分是外部利用系统机制带来的威胁；另一部分是系统级的安全机制本身。

外部利用系统机制的典型代表是 Webshell、木马、病毒、Rootkit 等，还有一些人为的安全问题，例如弱口令、缺少安全意识的服务配置和端口开放等。这些也都需要专门的技能进行检查，也有很多关于这方面可以使用的工具。

系统级的安全是不针对特定的攻击类型的系统提供的机制。例如 UGO 文件权限，对于进程的能力限制 Capabilities；针对文件的，例如 mount 文件系统，在针对文件时指定 ACL 就可以针对文件进行访问控制，还有一些程序级的组件一般会随着操作系统提供，这些组件也有一些是与内核耦合的。例如内核的安全框架 LSM，以及在 LSM 下实现的各种防火墙，实现 Flask 框架的 Selinux 和 Apparmor。Syslog 也属于内核提供的组件，可以用来做安全审计，但是其不是安全专用的组件。还有一些每个 Linux 发行版都会提供的基础应用组件，最常用的是 PAM 系统，PAM 系统的显著特点是可以把进程的认证工作由程序员转交给系统管理员，从而使得安全控制也变成运维的一部分。我们最常见的安全使用者的工作是权限配置，chown，chmod、chattr、chacl 等基础命令可以用来做权限配置。安全使用者更重要的工作是对系统不安全的地方打补丁，例如关注 CVE 和 Patch，检测自己的系统在不在漏洞影响范围，然后打补丁。得益于 Pam 和 Apparmor 良好的配置系统，安全系统的配置大部分都可以直接写文本配置，而不是实际的安全编程。此外，Netlink 可以查阅和修改更加详细的内核信息，audit 系统可以进行程序的追踪，这两者也是十分有用的技能。内核级别的安全工作对技能的要求相对较高，我们会需要 bpf 或者是 kprobe 钩子，懂得内核的细节技术，熟悉 ktap 和常用的模块开发和内核的入侵手法以定向检测等。

## 7. 内核使用者

内核的使用者多见于嵌入式开发和运维的内核升级。但是运维的内核升级一般涉及的功能裁剪较少，涉及的漏洞更新和功能增强较多。运维升级配置内核更重要



的是使用诸如 `zram` 等升级特性，并且强调不用重启机器的热更新。而嵌入式到目前为止还是基本停留在 2.6 的内核版本系列，最近几年博通等上游厂商已经升级到了 3.0 之后的内核版本。这里必须强调的一点是嵌入式行业使用的内核版本一般取决于上游厂商下发的 BSP（板级支持包）的版本，也就是说内核版本的迭代大部分由上游解决方案厂商控制。运维人员做内核是为了升级和定制内核功能，嵌入式做内核大部分是为了定制内核的细节。

很重要的一点是 Linux 内核从 2.6.32 版本之后，很难适用于低端的嵌入式系统。虽然内核仍声称为嵌入式应用做了诸多优化，但是业界的嵌入式开发基本停步在 2.6 版本的内核。你可以看到内核的新功能和增强基本都是为了互联网而产生的，而针对这样的内核进行嵌入式裁剪也越来越难，甚至要高版本的内核在嵌入式板子上跑起来这个基本的工作也越来越复杂。这也从侧面反映出了互联网的活力和嵌入式行业的守旧，也导致了嵌入式内核版本的升级权力被越来越多的掌握到职业做内核定制的上游厂商手里。

最基本的内核裁剪工作并不要求对内核是如何实现的内容有太多的了解，但是需要知道内核实现的那些功能有什么，为什么内核需要这些功能。例如裁剪中你得先选择 `net` 设备，然后 `net` 功能才是可选的，并且 `net` 功能里面的子功能非常多，一个看重 Flash 大小的嵌入式设备是不需要 `net` 功能下的大部分子功能的。内核的编译排版很重要的一点是按照功能的层级划分的，而不是按照功能的重要性划分的。比如你会发现无线系统里 RFID、LED、业余无线电和 Wi-Fi 是平级的，但是大部分人是不会使用前三者的，大部分人只需要 Wi-Fi。但是内核的编译选项的组织并没有针对这种需求上的流行程度进行优化。

所以一个内核裁剪者需要知道几乎所有内核选项的作用，最好多试试。内核的实现大部分为了通用性，对效率和安全的考量是非常少的。如果你深入内核的代码层次进行研究，你会发现内核的实现大部分在你深入使用的场景中，你会有更优的算法，你会想去重新实现。这里体现出的 Linux 内核的哲学就是：全面之后再求精。

所以内核会覆盖尽可能多的功能，但是大部分功能的实现都不是企业级的。例如，如果你的产品要支持打印机，你一般不会用内核内置的功能，而会购买更产品化的内核模块（例如 `Kcodes` 的打印机模块）。如果你的产品要支持 `Samba`，你会发现内核对 NTFS 的支持是非常低效的（在用户侧实现），还是会购买商用的 NTFS 内核模块（商用的和开源的是同一个公司开发的）。当你多关注内核的发展时，就会发



现开源发展得最好的模块一般是企业支持的，而在这背后一般都有商用版本存在。这就是内核的本质，出发点是开源的、共享的，发展是靠利益驱动的，繁荣则是完全靠商业的。

内核移植工作考验的大部分不是内核本身的技能，而是对 GCC 的了解程度，尤其是内核使用的 Makefile 系统。所以想要做好嵌入式内核的移植工作，编译系统和看一遍链接的 spec 是基本的要求。

### 1.2.3 开发者

开发者其实就是常说的程序员，而且一般是后端程序员。刚开始入门的程序员会更注重语法，只要是服务于实际线上业务的开发人员就有了解 Linux 的需求。不同种类的程序员的知识需求是不一样的。例如写业务代码的程序员，需要拥有架构能力和懂得编码标准；写高性能程序的程序员则需要拥有关于数学、算法和高性能编程的硬件相关的知识；写实时代码的程序员又是需要另外一套理论体系。编程的语法是基础，但编程的核心从来都不是语法。

选择了一门好的（计算机）语言，基本就能确定你要用它来做的事情。不存在“万金油”的语言，注重效率的和注重快速开发的、注重工程管理、注重描述问题的都不是同样的语言（当然你要用 C 语言做 Web 后台开发也可以），甚至还存在专门擅长处理字符串的语言。对于 Linux 来说，开发高性能代码一般就得是汇编，例如 C 和 C++。需要兼顾性能和开发效率时可以用 Golang，脚本化的语言也都可以用在 Linux 上，这取决于业务。但是为了工程的需要，例如当要求工程有快速性和易用性的需求时，Python、Erlang 也经常被用来做后端开发。

工程编程最重要的是库和架构，即代码的复用。一个成熟的程序员和一个入门级的程序员的最大区别不在于语法的熟练程度，而在于他们具有的架构能力和库的复用能力。内核使用的 C 语言看起来很容易，但是能写好的程序员基本要工作好多年，否则写的程序不稳定，可能存在安全问题，或不可读等项目管理的问题。

所以 Linux 对于开发者来说，不存在编程语言和库上的障碍。难点基本上是在内核所提供的功能上，以及你如何使用这种功能（利用 epoll、inotify 等）。

#### 1. 桌面应用开发者

Linux 下常见的桌面主要是 KDE 和 Gnome。从 Linux 桌面市场的占有率来看，

在 Linux 上面投资是不值得的，比如国内的常用软件也都很少会出一个 Linux 版本的。照目前看来，Gnome 占市场份额变得越来越大来的概率很大，不排除日后会有越来越多的厂商愿意为 Ubuntu 的 Gnome 桌面系统开发图形界面的应用。目前在 Linux 系统上的产品级的应用的图形界面一般使用 Qt、Java (swing) 等成熟的，可移植的图形库。

所以目前来看，如果你是桌面程序员（Android 系统除外），你可能要用 Java 或者 Qt 的 C++ 了，否则对于自身知识的积累可能会存在过时的风险。由于 Android 的 App 开发也是使用的 Java，所以目前最划算的选择是用 Java（企业应用市场对 Java 有强大的热情，因为它是最早普及的工业化的编码语言，但不代表它是目前最好的）。所以学习 Linux 桌面应用开发基本上就是学习这两款产品的相关知识。H5 和 javascript 也在比较快速的发展，但是目前对于桌面级别的开发并没有形成太大的冲击。

## 2. 使用成熟库的后端开发者

后端开发者占据了很大一部分开发岗位的就业比例，并且后端开发的难度可深可浅，但是变化速度不会太快。界面、网站等前端开发人数最多，变化最快，技术沉淀最难。几乎所有面向市场发布的程序都有后台服务器（单机程序是没有的），几乎所有的后台服务器都要存储数据。所以后台开发者要面对的核心开发点就是：网络使用、传输编码、数据存储、多线程编程和业务逻辑。所以做后端开发对自身的技术要求比较高，而目前的服务器几乎被 Linux “一统江湖” 了，这个趋势还会愈演愈烈。Windows Server 在发达国家应用比较广，但是 Linux 的免费和越来越成熟的稳定性，加大了对它的竞争力度。

最近 Golang 在后端开发的流行度迅速崛起，但是人们在大部分大型系统中还是选择使用 C 语言/C++。使用 Python 做后台开发的情况也有很多，也可以选择用 PHP 来做，但是在业内比较少见。由于相对新的高级语言大部分使用自带的网络库，所以本节就不涉及 Golang、Python 等语言的网络编程方法。

网络传输问题：网络常用的 C 语言/C++ 后端库有原生的 `epoll`、`libevent`、`libev`、`boost::asio`、`ACE`，`ACE` 一般产业界用得较少。`libev` 理论上比 `libevent` 高效，但是在实际使用时要视具体情况和使用者的使用方法。部分工业级的开发使用 `libevent` 或者原生的 `epoll` 相对较多，也有使用 C++ 的 `boost::asio`。但是这种情况比较少见，因为 C++ 难度高，而目前的现象是大部分网络基础服务是使用 C 语言开发的。

传输编码问题：以前是直接使用自定义的格式或者自定义的 Json，后面加压缩



来作为传输消息的载体，后来发展出了序列化。再后来序列化进一步发展形成了 ProtocolBuffer、Thrift、Avro 等大公司主导的传输格式。目前网络传输数据用得相对多的是 ProtocolBuffer，因为有谷歌的“背书”，Thrift 也在强势崛起。Avro 则刚刚起步，但是特性不俗。很多工程负责人喜欢成熟的可以直接使用的消息队列组件，但这些消息队列的下层也是要在传输的格式上进行不同的选择。

**数据存储问题：**MySQL 数据库几乎是大小系统的第一选择。非常小的系统可能会使用 Sqlite，涉及非 IT 大型企业可能用商用的数据库比较多，Nosql 里 Mongodb 被选择的频率比较高，它也是最接近关系型数据库的，类似 HBase、CouchDB 等相对常见的非关系型数据库也都风靡一时。近年也出现了很多特定用途的 Nosql 数据库，一般在实际的线上工程中在选择上都会相对谨慎，即使被选数据库与我们的需求恰好相符。例如有专门存放图的数据库（ArangoDB），也有分级存储数据内容的（Rocksdb），还有存储地理信息的，等等。如果你是专业方向的开发者，可能这些（专门的）数据库更适合你，但是也得始终持有谨慎和怀疑态度。

**多线程编程，**在 C 语言/C++的世界里没有太多的选择，大部分情况下使用 pthread，C++可以用 boost::thread 或者 C++11 的 thread，其后台也是 pthread。pthread 在 Linux 平台基本可以说是目前在工程上的唯一选择。

所以使用程序库的开发者只需要了解库的用法。当然更希望他们了解库后台是如何调用操作系统，如何具体实现的。boost::filesystem 的很多函数使用系统的函数也并没有太大的难度。事实上，我们总是可以使用 lstat 函数来替代 boost::filesystem 的一系列文件存在性和类型的判断函数。因此 boost 很多库确实有过度封装的嫌疑，尤其是你只需要一个特定的平台时，例如值在 Linux 下运行。使用 boost 有时还需要单独考虑 libc\_nonshared.a 中对 stat 系列调用的定义。但是其封装的，例如 remove\_all 等方便的遍历删除和 path 这种方便操作的类定义还是有比较大的价值的。即便如此，熟习底层的程序员仍然会选用 ftw 这种函数调用。

### 3. 系统级后端开发者

如果你打算看看你的 Linux 发行版上已经安装的库是用来干什么的，例如 libncurses、libnss、libfuse 等，而这些库在你平时开发应用程序时都用不到，那么你要做的就是系统级的后端开发。系统开发与操作系统的关联很大，学习系统开发就是在学习操作系统。

在系统开发时对内核信息的获取要通过 Proc、Sys 和 Netlink，这是一定要熟练





掌握的。Proc 相对容易，但是内容很多。Sys 则比较庞杂，内容会比 Proc 更多。例如你得清楚地知道 `/proc/sys/kernel/core_pattern` 里面存的是 Core Dump 的路径；`ulimit -c` 可以用来设置 Core 文件的大小，默认是 0。这些基础的背景知识以及整个文件系统衍生出来的知识点是系统级后端开发的基础。常用到的 Netlink 都会有对应的上层封装，但是封装的质量可能不理想。例如 `rtnetlink` 这个最常用的子接口最好自己学会编写，但是如 Audit 系统的 Netlink 就可以直接使用 `libaudit`。

这些常用的开发内容包括但不限于：FIFO 文件、`uevent`、`inotify`、Netlink、`nice`、`cpu` 亲和度、`cgroup`、虚拟化、`ptrace` 进程跟踪、子进程创建和控制、信号处理、文件锁、向量化的读写文件、文件描述符操作、`socket` 调用、`epoll`、文件与目录链接控制、锁、磁盘配额校验、进程记账、权限控制、运行优先级、低级端口操作、`sg` 直接发 SCSI 命令、交换分区控制、`pdflush` 和 `kswapd` 等内核进程的调优、模块的装载与卸载、`mmap` 和 `brk` 的内存映射、`cache` 操作、网络设备操作、用户管理、消息队列、信号量与共享内存等。

系统级后端开发直接是面向内核的使用，也就是系统级后端的开发者基本就是内核的合格使用者。

#### 4. 运维开发者

运维开发者比较接近于系统开发者，但是运维开发者比较多的情况是使用现有命令、脚本，着重于系统资源的监控和划分。现在流行的 `devops`，例如 `ansible` 工具让运维与开发一气呵成。运维开发者首先是一个运维使用者，运维系统，例如全网监控系统、包发布系统、主机探测系统、域名系统等都是运维开发者的发展方向。一个运维开发者不做具体业务，也不是直接为具体业务服务的，他是让具体业务可以专注于具体业务的。

#### 5. 安全开发者

防御型的安全开发者有两种，一种是如何让自己开发的软件更安全；另一种是开发安全防护软件，例如病毒扫描、防火墙、入侵检测、漏洞管理、权限控制等。除了对安全使用者的技能的掌握外，还需要更深入地了解白帽子们的安全防护细节和原理。通常能“防”的人也能“攻”，不知道别人怎么攻就在防方面基本做不深入，攻防是互动进步的。但是目前国内的安全系统大多数是由专门做安全防御的人做，很可能他们根本没有做过实际的攻击。主要原因是国内的网络安全环境太好了，基





本在做坏事的都是“脚本小子”或者使用工具进行攻击的人，拥有特别高攻击水平的黑客的确存在，但是目标也基本不会是一般的防御系统，并且他们的入侵对于小企业都是无感知的。

内核里有很多针对安全开发的特性提供：内核加密接口和秘钥环、ASLR（进程启动栈随机化）、LSM 机制。做安全开发对系统本身的特性利用不大，对攻防手法的理解要求比较大，例如对 ELF 格式的深入理解就可以在二进制加固与破解领域大展身手。另外，防御系统一般是在业务系统的前面，所以要求低延时和高吞吐。所以基本上只能使用 C 语言/C++，很难想象前端过滤的防火墙用 Python 写是什么样。

所以安全开发的核心是业务和高效编程的能力。而高效编程，例如对 DPDK、SSE 指令集的使用就是一个专门的学科了，对业务上的要求就是需要掌握安全相关的知识点。安全编程更多的在于领域知识的掌握。相比于需要掌握的领域知识，学习 Linux 提供的安全机制是相对简单的。

## 6. 应用后台开发者

对于大部分应用后台的要求有两个：开发快，问题少。所以现在的市面上你会见到很多用 Golang、Python、Java 做后台开发的案例。这种形式的后台开发基本与操作系统无关，懂得基本的 Linux 系统的使用即可，人们可以专注地面向业务。

## 7. 内核开发者

对于内核开发者的技术要求是最高的，如果谁提交了一个 Patch 被内核接受了，那是很了不起的事情。因为内核本身进展就非常大，并且内核开发在市场上没有对应职业。最多的相关职业是驱动开发和内核裁剪小修改，另外文件系统开发和网络开发也涉及一些内核开发。由于内核的庞杂和耦合性比较重，学习本身就很难了，更别说开发了。

但是内核开发入手是相对很简单的，只是在实际的环境中缺少对应的职位，并且即使存在内核开发的需求也是使用的很老的内核版本进行稳定性开发，几乎没有机会可以去写一个新的机制。实际的内核开发的门槛是相对很高的。这里的开发者有的是开发内核机制的；有的是开发驱动的；有的是开发文件系统的，针对不同的开发有不同的知识侧重点。

驱动开发对 uevent、kobject 系统的了解需求比较多，明白 udev 程序和 dev 目录的工作原理、设备号的管理、基本的内存申请和使用。进阶的学习可以了解内核端



socket 编程、进程的控制等。内核驱动的编程最主要的还是业务逻辑，要知道你控制的设备的寄存器和对应的总线在内核中的逻辑，例如所有磁盘都使用 SCSI 命令，都要经过 SCSI 层。这时对 SCSI 的了解就显得比较重要了。

由于一切皆文件的思想，所以文件在 Linux 中特别重要。如果想要创建虚拟的设备，得学会利用/dev 目录下的设备，重要的是你得学会使用 fd 文件句柄。这个 fd 就是简单的 C 语言里面 open 一个文件之后生成的返回值，但也是 socket()调用之后生成的返回值，这个 fd 的值是内核分配的，你不能设置打开资源的请求获得的 fd 的值，但是可以推测。由于是系统资源，所以 fd 实际上是跨进程的，但是例如 socket 生成的 handler，虽然是系统资源，但是其他进程就不能直接使用，因为这个系统资源有个所有者的概念，这个所有者就是创建它的进程。像 0、1、2 号的 fd 就是输入输出和错误，这个 fd 是在 proc 文件系统下可以看到的。由于子进程可以得到父进程的所有 fd，所以通过 shell 其实可以做好多事情。例如常用的 exec 命令就是利用 fd 的典型方法。如图 1-1 和图 1-2 所示。

```
root@ubuntu:~# ls -l /proc/1171/fd
total 0
dr-xr-xr-x 2 root root 0 Apr 27 23:22 /
dr-xr-xr-x 9 root root 0 Apr 23 22:44 /
lrwxrwxrwx 1 root root 64 Apr 27 23:22 0 -> /dev/null
lrwxrwxrwx 1 root root 64 Apr 27 23:22 1 -> /dev/null
lrwxrwxrwx 1 root root 64 Apr 27 23:22 2 -> /dev/null
lrwxrwxrwx 1 root root 64 Apr 27 23:22 3 -> socket:[16510]
lrwxrwxrwx 1 root root 64 Apr 27 23:22 4 -> /var/lib/dhcp/dhclient.ens33.leases
lrwxrwxrwx 1 root root 64 Apr 27 23:22 5 -> socket:[16549]
lrwxrwxrwx 1 root root 64 Apr 27 23:22 6 -> socket:[16550]
```

图 1-1

```
root@ubuntu:~# lsdf -p 1171
lsdf: WARNING: can't stat() fuse.gvfsd-fuse file system /run/user/116/gvfs
Output information may be incomplete.
COMMAND  PID USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
dhclient 1171 root   cwd  DIR    8,1      4096        2 /
dhclient 1171 root   rtd  DIR    8,1      4096        2 /
dhclient 1171 root   txt  REG    8,1     487248 393905 /sbin/dhclient
dhclient 1171 root   mem  REG    8,1     47600 262625 /lib/x86_64-linux-gnu/libnss_files-2.23.so
dhclient 1171 root   mem  REG    8,1     14608 262226 /lib/x86_64-linux-gnu/libnsl-2.23.so
dhclient 1171 root   mem  REG    8,1    1864888 262340 /lib/x86_64-linux-gnu/libc-2.23.so
dhclient 1171 root   mem  REG    8,1     405432 270799 /lib/x86_64-linux-gnu/libisc-export.so.162.0.0
dhclient 1171 root   mem  REG    8,1    1917280 270793 /lib/x86_64-linux-gnu/libdns-export.so.162.1.0
dhclient 1171 root   mem  REG    8,1     162632 262390 /lib/x86_64-linux-gnu/libidn-2.23.so
dhclient 1171 root   0u   CHR    1,3        0t0        6 /dev/null
dhclient 1171 root   1u   CHR    1,3        0t0        6 /dev/null
dhclient 1171 root   2u   CHR    1,3        0t0        6 /dev/null
dhclient 1171 root   3u   unix  0xffff880075985400 0t0 16510 type=DGRAM
dhclient 1171 root   4w   REG    8,1     1467 529813 /var/lib/dhcp/dhclient.ens33.leases
dhclient 1171 root   5u   pack  16549    0t0    ALL type=SOCK_RAW
dhclient 1171 root   6u   IPv4   16550    0t0    UDP *:bootpc
```

图 1-2

这个文件列表比较典型，fd 既有设备的映射，也有 socket 的映射，也有文件的映射。可以发现，如果不希望使用输入输出，典型的是不希望看到输出，这里的 1



号和 2 号 fd 就可以被重定向到/dev/null 设备文件中。例如你可以使用 Shell 命令 `exec 1>outfile`, 就可以直接把当前打开的 shell 的输出重定向到 outfile 文件中, 这种方法用于脚本中重定向当前脚本的 fd。

内核中文件系统的开发, 必须得了解文件使用的整个流程, 一切皆文件的思想使得最上面的接口必然是 VFS 层, 往下还有通用块层 (在这里要进行重要的电梯算法)、SCSI 层, 要把对逻辑文件的访问变为对物理存储访问的命令, 还要经过 PCI 层, 如果是 USB 设备还要经过 USB 层。逻辑最终成真总要经过物理设备, 所以了解物理协议的实现也是必要的。

内核中提供了很多默认的文件系统操作, 很多实现的文件系统都直接使用默认的实现。有一类重要的文件系统是 Fuse, 文件系统驱动可以使用这个机制在用户端实现, 像个普通的应用程序一样。这是内核为版权保护做的折中让步。内核现在越来越倾向于把功能让给用户空间, 大内核的思想在收缩。

文件系统一般可以以模块的方式提供, 可以很简单也可以很复杂。所以文件系统开发对内核的了解与其他的内核功能开发差别不大, 但是对文件系统本身有比较高的知识储备要求。例如完整性校验、Extends 大块、磁盘配额、磁盘访问控制 ACL、热插播、B+树等。

## 8. 网络开发者

Linux 内核本身的网络协议栈相对低效 (比如对比 6WIND), 但是可以应付绝大多数的使用情况。希望使用内核原生的协议栈做动作修改的用户, 一般是使用 Netfilter 的 Hook。我们可以使用内核的模块做很多事情, 但是对内核代码本身的修改在工程中是不建议的。具体做安全还是包变换, 很多时候 Netfilter 的 Iptable 本身就可以做, BPF 更是提供了可编程的规则, 所以内核层面的网络开发核心是 Netfilter。

对于有高性能网络数据处理要求的情况, 有新浪的 Fastsocket 和 Intel 的 DPDK 这两种常用的协议栈可以供选择。Fastsocket 目前对长连接的支持说不上完善, 但是 Nginx 这种短连接应用会从中收益良多。DPDK 没有 socket 的概念, 是纯粹的包处理, 而且是在用户空间, 完美地支持多 CPU 和 NUMA 系统。所以可以看到阿里、腾讯、谷歌、百度等都有用 DPDK 来做的数据包处理。

网络开发对于开发者的要求也大多数是业务算法上的, 也不是内核的实现上。DPDK 几乎是完全绕过了内核的网络实现, 使用单独的接口 (除了与内核的网络桥接 KNI 设备)。



## 1.3 如何形成一个内核

Linux 内核并不是唯一的内核，也并不是唯一的一种内核。实际上，Linux 内核属于 Monolithic Kernel 的一个实现，这种内核还包括 UNIX 系列（BSD、SunOS 等）、DOS 和 Windows 9x 系列，还有 OpenVMS、XTS-400、z/TPF 等一些不常见的系统内核。

从学术上看内核的种类包括 Exokernel、Nanokernel、Microkernel、Hybridkernel、Monolithickernel 和 Anykernel 这 6 种。Windows NT（9x 之后的所有）都属于 Hybridkernel，因为它系统功能的一部分在内核中，一部分在用户空间中。

内核提供的核心功能只有一个：资源管理。资源管理大致分为 3 种，即 CPU 资源调度、内存管理、I/O 设备管理（内存管理也归在此类）。

### 1.3.1 内核形成过程

我们分析一下在一个裸板上从头写一个操作系统要经过的过程。

首先，我们肯定要把汇编的启动代码封装一下，然后写一些启动的 C 代码，内存需要汇编来初始化，才能向其中加载内容。但这也并不是绝对的，我设计过的一个作品就是一个裸片的专用操作系统的实现，但是并没有使用任何的汇编，而是将要操作的寄存器地址直接定义为变量，你能在很多嵌入式系统中看到类似的定义。

加载之后，我们可以在内存中执行代码了。但是此时内存中是无章法的一大片连续块，有的平台甚至还不连续，物理地址具体会被映射到什么样的线性地址，还需要查看各个芯片的手册才知道。很多芯片还有特殊的映射结构和启动逻辑，例如三星的 ScurityZone，我们在启动的早期必须要具体问题具体对待，这也是各种各样驱动存在的原因。要使用 C 语言就必须为内存划分块（全局区、代码区等），我们可以让 GCC 去完成这个工作，但是通常需要手动布局一下。当涉及堆，很大的内存时，就需要考虑内存分配算法和页的划分了。有了页的划分就得有缺页中断的实现了。这部分就实现了内存管理。一般来说，用 GCC 作为主要工具来写操作系统的时候，只需要实现堆的内存分配算法。而如果是裸片程序，堆都可以不用实现，直接使用一大块连续的内存，自己在内存中组织自己的数据结构就可以。事实上，在很多嵌入式开发的时候都是强制要求在运行过程中不得使用堆内存的，而是在程序





执行的开始时分配大块内存自己用。

然后我们需要初始化外部的设备，由于外部的设备都是寄存器控制的，寄存器通常是功能复用的，通过写入功能号来执行功能，通过查看一些寄存器的位来确定状态。如 PCI 这种总线硬件更复杂，还会有单独映射的配置空间，网卡有相对通用的 MII 中间层。针对一类硬件的访问就需要封装，而分类类型可大可小，例如我说写入到存储设备是一类操作，但是存储设备也分为 SSD 和磁盘，磁盘又可以细分为很多不同的类型。这样逐层封装就形成了硬件抽象层。如果同样是鼠标，不同的商家的寄存器排列不同，那么我们就得针对每一种鼠标实现一个驱动，然后驱动的上层接口与硬件抽象层相符。于是驱动和硬件抽象层的概念就产生了。

如果到此为止，只实现了基本的硬件管理和硬件抽象层，“Nanokernel”就产生了。而一个操作系统可以多进程，进程间还要通信，各个进程之间还要调度。内核里实现了进程概念这一步，就是“Microkernel”。再实现一些文件系统、网络协议栈等就是“Monolithickernel”了，如果在用户端也实现了一些系统级功能，就是“Hybridkernel”了。

### 1.3.2 Exokernels 和 Anykernel

Exokernels 是最微小的内核，其功能仅限于限制对资源访问进行复用和保护。对硬件做最基本的抽象，允许应用无限权限的访问硬件。也就是说，Exokernel 只是为应用提供一种最小化的硬件抽象，怎么使用是使用者说了算，也就根本不知道进程这种概念了。

NetBSD 实现了第一个 Anykernel (Rump Kernel)。Anykernel 本身既是内核又是程序。作为程序它向基于它的其他程序提供内核的功能。作为内核，因为它本身就包含内核的功能。也就是说，Anykernel 在任何操作系统上是可移植的。例如在 Linux 系统上可以使用 Rump Kernel 的 TCP/IP 协议栈，下层直接用 DPDK，上层的应用调用 Rump Kernel 就可以完成 socket 协议栈的操作。这种方式本质是一种虚拟化，与其他的虚拟化方式不同的是，其他的虚拟化虚拟的是硬件，希望运行不同的软件，而 Anykernel 的思路是虚拟软件，可以运行在不同的操作系统上。Rump Kernel 的实现包含了文件系统、协议栈和设备管理。



### 1.3.3 内核为何使用 C 语言

关于 C 语言与 C++，大部分认为 C 语言执行效率高。但很多人做过实验，如果 C++ 不使用 RTTI，C++ 的效率也不会比 C 语言的效率低太多（25% 左右）。还有人说 C++ 虽然拥有强大的 STL，但是对于极度追求效率的情况，STL 不能用。大部分人的心态是，学 C++ 出身的就经常埋怨 Linux 的 C 语言代码乱的一塌糊涂，崇拜自己的各种敏捷，面向对象原则，代码不如 C++ 精简，连 STL 或者 Boost 都用不上，还有软件工程相关问题都是被他们抱怨的“重灾区”，但是这部分人大多数都没有长时间地写过 C 语言。

面向对象的实现方法不止 C++ 一种语言，C 语言也可以做到，只是不如 C++ 那样浑然天成。重要的是 C 语言做的面向对象，由于封装性差，一个大型的结构体里面什么都有，看起来比较混乱。虽然可以用多层次的结构体定义在一定程度上让代码更加可读，但是内核中大部分情况下并没有这么做。而用 C++ 封装的结构体，可以设计得很潇洒，`vector<People>` 会比 `list* people` 能传递更多的信息。利用 C++ 的封装特性带来的好处在视觉上显而易见，基本不需要考虑内存布局，但在 C 语言的结构体定义中到处都有对内存上的考虑，例如通过结构体成员找到结构体本身的 `container_of`；在结构体最后添加一个 0 长度的数组；在结构体的开头添加一个 `list*` 节点。乍一看 C++ 也能做到，毕竟 C 语言的所有关键字 C++ 都是完全支持的。但是问题是这并不是面向对象的思路，有一种编程思路叫作 ODP（面向数据编程），就是典型的 C 语言更加合适的用途。说 C 语言比 C++ 高效，并不一定是说 C++ 本身效率低，而是其所代表的面向对象的思路执行效率低。举个例子，代码如下。

```
struct A{
    int a;
    int b;
};
vector<A> aList;
```

我们对这个 `aList` 的 `a` 变量进行遍历，比起下面这个写法，代码如下。

```
struct A{
    vector<int> a;
    vector<int> b;
}aList;
```



一定是下面这个写法的效率高，原因在于缓存。所有的计算机系统都有缓存行，第二种写法被调入缓存的全部是 `a`。而第一种写法还调入了 `b`，遍历起来第一种写法比第二种写法，是将缓存减小了一半。这就是 ODP 比 OOP 高效的地方。而当你不选用 OOP 时，C++ 比 C 语言多的最大优势就荡然无存。

C++ 的确有强大的 STL 和 Boost 库。但是这种库都是通用的，大型的企业用到后期对执行效率开始敏感的时候，通用的东西几乎不可能在所有的情况下都是最优的选择。包括内存算法、进程调度、网络应用，简单地说就是要针对不同应用调整参数，例如 TCP 在服务于 Samba 时要关闭 Nagle 才能提高效率，但有的应用 Nagle 则是提高效率的利器。而 C++ 的 STL 给与我们对库运行策略调整的能力过小。内核是底层的系统，使用这种程度上通用的东西确实是不现实的。最通用的数据结构应该是 list 了，在数据组织上也是非常轻量的，甚至在服务于不同的应用时还有 hlist 和 llist 等变体。这种变体对效率的追求不是 C++ 的再封装所能满足的。

C++ 最骄傲的特性是：封装、不过于关心内存、虚函数、继承。虚函数确实是在机制上就存在效率问题，一用它就得多一个虚函数表，如果这不算是小事，函数调用跳转带来的缓存的刷新可真不是件小事了。我们只要用了虚函数，很难控制哪个地方就刷一下缓存，这对于内核来说是不可接受的，因为内核要实现一个算法的时候都要小心翼翼地关心每个函数调用的操作是不是性能上潜在的一个陷阱。至于继承，一个写了很久 C++ 代码的人都有一个体会：除非是做大型应用软件，否则业务的建模真没那么多变体。对于内核开发来说，虚函数基本就是可有可无的鸡肋，可能会有几个比较重要的大型功能框架可以用到这个特性，典型的是驱动体系，但这完全不构成因此而选择这门更复杂语言的理由。如果我们只是开发大型商用平台式应用软件，恐怕都要考虑内存布局，甚至是有的效率敏感代码还要刻意挑出来用 C 语言或者汇编重写。

我们在使用 C++ 的时候有很多设计模式，有很多编程技巧。它们带来的效果大部分是架构清晰、代码量变少、易于修改。但是 C++ 带来的更多的是内存的使用量飙升和代码体积的增大。C 语言的代码确实多一些，但是过程式的代码比起 C++ 要到处去找到到底是哪个虚函数或者哪个子类在起作用要方便得多。毕竟 C++ 有的是运行时才确定的，而我们看代码的时候是静态的。这有一个本质的原因：C 语言可以看静态代码得出编程的全部意图，但 C++ 不能或者很难。而内核的开发者真正要动手写的非常少，几乎都是稍微改改代码或者是编写一下驱动。所以对于内核来说，即使是深度使用内核工作的人，大部分也是使用者，而并非开发者，那么追求敏捷





的意义又有多大呢？

笔者也曾诟病内核的很多 C 语言代码写的比较混乱。比如电梯算法的框架代码，得非常认真地长期阅读才能大概体会作者的意图。然而我知道，即使我很容易看懂了电梯算法的代码，我也不会修改它。它提供了很多参数，然而大部分情况下我通过调整参数就可以满足自身需求。即使在全球范围内，真正对于重写这块代码或者对写自己的电梯算法有需求的公司恐怕都屈指可数，修改和维护工作自有作者自己在做。如果一个员工告诉上司他修改了很多电梯算法的框架时，公司的领导估计是不敢用的。做到这个程度的产品基本是嵌入式产品，嵌入式产品的软件一发布就不容易更新软件了。如果可以通过修改参数来完成，怎么会有决策者采用冒险的修改内核代码的方式呢？毕竟技术在产品面前，是要服务于产品的。

笔者也是 C++ 的拥护者，甚至是狂热者，日常的软件、大部分的公司产品开发都会选择 C++ 或者 Java，而绝对不会选择 C 语言。然而在深入学习内核功能之后，如果可以重新选择内核使用的语言，笔者也会选择 C 语言。虽然笔者也曾经用 C++ 写过裸片操作系统，也长期使用过 eCos。面向对象确实是编程的一大进步，那我们也得思考一下数据库大行其道的为什么不是实体联系模式，而是关系模式呢？我们可以采用炫酷的技术，因为技术确实推动了生产力革命，当然也不能迷信技术，工程和技术是完全不同的两回事。





## 2

## 第 2 章

## 内核架构

虽然本书不是讲解内核的核心原理，但是如果不学习内核相关知识，对于本书其他知识点的理解也会变得困难，所以从通体上观察内核的架构对于辅助理解其他知识点是十分有必要的。

## 2.1 常见架构范式与核心系统

在 Linux 内核中使用了很多“约定俗成”的架构上的约定，这些约定大部分被 C 语言开发者所熟知，并被广泛应用到日常的使用 C 语言开发的应用中。

### 2.1.1 Linux 内核上下层通信方式

定义一个结构体，包含各种函数指针并且管理其列表。下层通过生成一个这样的结构体，将自己的操作函数赋值给该结构体的对应域，然后调用上层的注册函数，将自己的信息注册到上层。这样上层就可以用统一的函数调用不同的下层接口。上层可以遍历，通过名字或设置哪个函数有效来确定调用哪个下层结构体对应的同名操作。这种方式使用了 C 语言，同时融入了面向接口和面向过程的编程思路。一个典型的例子如下。

```

struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control
*);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,
        struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
        loff_t pos, unsigned len, unsigned flags,
        struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
        loff_t pos, unsigned len, unsigned copied,
        struct page *page, void *fsdata);
    sector_t (*bmap)(struct address_space *, sector_t);
    void (*invalidatepage)(struct page *, unsigned int, unsigned int);
    int (*releasepage)(struct page *, gfp_t);
    void (*freepage)(struct page *);
    ssize_t (*direct_IO)(struct kiocb *, struct iov_iter *iter, loff_t
offset);
    int (*migratepage)(struct address_space *, struct page *, struct page
*, enum migrate_mode);
    int (*launder_page)(struct page *);
    int (*is_partially_uptodate)(struct page *, unsigned long,
        unsigned long);
    void (*is_dirty_writeback)(struct page *, bool *, bool *);
    int (*error_remove_page)(struct address_space *, struct page *);
    int (*swap_activate)(struct swap_info_struct *sis, struct file *file,
        sector_t *span);
    void (*swap_deactivate)(struct file *file);
};

```

例如这个结构体（linux/fs.h）就是定义了一些约定的函数指针。当上层调用一个下层的函数时，一般有 3 种返回值的方式，第 1 种是通过语言规定的函数返回值（在大部分情况下）；第 2 种是传递的参数是指向结果的指针（例如 writepage 函数的 wbc 参数）；第 3 种是使用回调函数（在内核中有广泛用处），通常让用户提供执行过程的某一处逻辑甚至是用户定义的返回结果。回调函数通常是调用者提供的。

内核首先按照功能分成了几个大型的子系统，再通过上下层通信的方式将内核

各个子系统有效地分成了层级关系，使得架构清晰。虽然这个手法并不一定是 Linux 首创，但后续无数的 C 语言程序员都受到这种“风格”的影响，可以说是 C 语言中的软件工程“导师级”的作品。

### 1. 资源竞争处理

内核的主要工作就是安排资源，而 Linux 是支持多进程的，这对资源竞争的处理就不可避免。内核在架构级别的资源竞争处理的主要思路是队列缓存，例如将磁盘读写产生的 bio 结构体提交给设备，当其发现设备忙时就将该请求加入队列，然后返回，如果设备不忙就直接提交。实现方式比较有趣，利用对象的队列指针判断设备是否繁忙。如果队列指针为空，则说明设备不忙。如果设备忙，该指针则不为空。

针对细节的资源竞争，Linux 提供了很多的锁的实现。如果 Linux 出现了不能满足需求的低效事件，那么基本上都是由于这些锁导致的。这些锁在第 5.3 节中有比较详细的介绍。

### 2. Linux 内核的结构体

Linux 的结构体之间可以有继承关系，但不是 C++ 那种显式的继承，而是子结构体包含了父接口体的指针，通过子结构体可以找到父结构体。而子结构体可以继承多个父结构体，就像 C++ 里面所说的强引用关系。示例如下：

```
struct A{  
}  
struct B{  
    struct A* a;  
}
```

总体来说，Linux 有两种结构体模型，一种是对对象的建模；另一种是对操作的建模。两个体系互相有交叉关系，又各有各的继承结构。

Linux 很多重要对象的结构体中的域都是为各个层次的各个功能使用的。开发者在写代码的时候出于一定的理由，需要在结构体提供某个域以完成自己的功能，这就会在其中找地方添加，而这个添加通常导致不同层级的需求被糅合在一个大型的结构体里，最后导致的结果就是非常难懂，得熟悉了组成结构体的细节组件之后才能有一定的理解。所以看 Linux 的结构体必须要对应细节功能，而一个结构体中的功能的跨度可能很大，所以看这种代码学习曲线很高，必须要全面地理解内核后才能看懂部分代码，所以需要迭代渐进地阅读和学习。



## 2.1.2 横向系统和纵向系统

横向系统不是指具体的物理功能，而是指各个功能模块都需要对外展示的接口。例如 `cgroup`、`proc`、`sys` 文件系统、系统调用的组织、调试系统、`Core Dump`、信号、内存管理等，以及内核提供的基础元素功能，例如 `workqueue`、`tasklet`、`pipe` 等。

在使用 Linux 时，横向系统是通用工具意义上的存在，无论研究 Linux 的哪个方面，都应该掌握横向系统，但是横向系统也没有绝对的划分。

纵向系统是指具体的功能模块，这些模块总体像是一个森林，不同的功能树下面有很多的层次。例如一个用户端对 USB 文件的操作要走完内核中的很多个层次，即文件系统层、缓存层、通用块层、SCSI 层、USB 层，每个层次的内部又分为多个层次。但 Linux 内核一般将它们分为 3 个层次，为上层提供统一接口的接口层，中间实现主要逻辑的功能层和统一下层不同驱动接口的驱动层。

不同层次有不同分工，它们按照顺序完成工作，一整条链路下来就是 Linux 的一个纵向的子系统。例如文件系统层解释文件在磁盘中存储的物理布局；缓存层实现读写请求在内存中的存放方式（预读、延迟写等）；通用块层处理上层多个请求的调度（例如合并、排序）；SCSI 层负责把上层命令的内容转变为设备可以识别的 SCSI 指令；USB 层负责把指令成功地送达给对应的设备。

使用 Linux 作为后台开发系统的开发者大部分情况是使用 CPU 的运算能力和网络的联通能力。而嵌入式开发者可能会关注 Linux 的热插拨服务，有的则会格外重视电源管理等。

很多 Linux 内核提供的服务都是互相交错的，所以横向与纵向的界限并不明显，这里的区别只是存在于对概念理解上。

## 2.2 基础功能元素

### 2.2.1 模块支持

#### 1. 模块概述

模块是 Linux 支持动态功能扩展的最主要机制。内核代码中有很多模块，如果

有了当前使用的内核的代码树，那么用户也可以编写外部的模块，动态添加到内核中执行即可。但是 Linux 内核的主要代码是遵守 GPL（GNU General Public License）协议的，其内部暴露给模块使用。如果用户在内核模块进行编程时要使用内核内部定义的调用，就需要将自己完整的 GPL 公开，这就是 GPL 的传染机制。

有很多公司，例如做 NTFS 文件系统内核模块驱动的公司 Tuxera，其最著名的产品是用户端开源的 NTFS 文件系统驱动 `ntfs-3g`，然而这个驱动的效率并不高。不是公司没能力优化，而是不愿意公开，另外它们还提供闭源版的 NTFS 内核模块驱动，如果要使用则需要购买。再比如 Kcodes 公司的打印机模块，该模块可以识别和处理几乎所有的打印机。虽然该模块的功能非常强大，但是也是闭源的，也需要购买。

那么为什么这些公司可以闭源而不必遵守 GPL 协议呢？原因是它们大多数都采用两个模块的方式来规避，一个是 LGPL 模块，封装内核的 GPL 调用，但是自己本身是 LGPL 协议的，另一个模块则可以直接调用在第一个模块中封装过的，只有 LGPL 要求的系统这种方式严格的说也并不符合 GPL 协议的要求，但是在 Linux 内核中却是被允许的。GPL 协议的传染性不能穿透二进制（否则用 GCC 写的所有程序都得开源）。

模块的执行原理与其他功能组件一样，都是模块开发者实现约定好功能的函数，然后使用规定的函数注册，在添加、关闭模块的时候内核模块调度系统就会执行用户注册的自己实现的函数。在模块开发中比较典型的是初始化和退出函数。

```
static int __initinit(void)
{
    printk("Hi module!\n");
    return 0;
}

static void __exit exit(void)
{
    printk("Bye module!\n");
}

module_init(init);
module_exit(exit);
MODULE_DESCRIPTION("Hello World !!");
MODULE_AUTHOR("liujingyang");
MODULE_LICENSE("GPL");
```

以上代码就构成了一个模块。可以看出 `module_init` 和 `module_exit` 就是注册约定函数的调用。模块内部定义的钩子函数是 `static` 的，目的是只内部可见。`__init` 和 `__exit` 是 GCC 的特性，提供回收明确表示无用代码的能力。标记为 `__init` 的函数会被放入 `.init.text` 代码段，这个代码段在模块加载完后会被回收节省内存，因为不会再用。最后的三个宏在很多版本上也可以编译和加载，但是一般尽量添加，尤其是在严肃的工程开发中。比如缺少 `MODULE_LICENSE` 宏的版权声明，在加载的时候内核就会告警：“Warning: loading hello.ko will taint the kernel: no license”。

## 2. 内核符号表

内核符号表是内核内部各个功能模块之间互相调用的纽带，各个模块之间依赖这些函数调用进行通信。各个功能模块必须要导出符号表才能被模块使用。还有动态加载的模块的链接需求，在加载时符号表是对内核其他部分描述本模块的最好方式。加载的模块所导出的函数通过导出操作就可以被其他模块定位并调用。示例如下：

```
static struct request *blk_old_get_request(struct request_queue *q,
intrw, gfp_tgfp_mask)
{
    struct request *rq;
    BUG_ON(rw != READ && rw != WRITE);
    create_io_context(gfp_mask, q->node);
    spin_lock_irq(q->queue_lock);
    rq = get_request(q, rw, NULL, gfp_mask);
    if (IS_ERR(rq))
        spin_unlock_irq(q->queue_lock);
    return rq;
}

struct request *blk_get_request(struct request_queue *q, intrw,
gfp_tgfp_mask)
{
    if (q->mq_ops)
        return blk_mq_alloc_request(q, rw,
(gfp_mask & __GFP_DIRECT_RECLAIM) ?
0 : BLK_MQ_REQ_NOWAIT);
    else
        return blk_old_get_request(q, rw, gfp_mask);
}
```



```

}
EXPORT_SYMBOL(blk_get_request);
void part_round_stats(intcpu, structhd_struct *part)
{
    unsigned long now = jiffies;

    if (part->partno)
        part_round_stats_single(cpu, &part_to_disk(part)->part0, now);
    part_round_stats_single(cpu, part, now);
}
EXPORT_SYMBOL_GPL(part_round_stats);

```

以上是摘自 block/blk-core.c 的 3 个函数，blk\_old\_get\_request 是内部使用的，blk\_get\_request 使用了 EXPORT\_SYMBOL，可以被任何其他的模块和内核部分使用（仍然需要遵守 GPL 开源），而使用了 EXPORT\_SYMBOL\_GPL 的 part\_round\_stats 函数则是只能被声明自己是遵守 GPL 协议的模块使用。EXPORT\_SYMBOL\_GPL 和 EXPORT\_SYMBOL 的区别在于每个模块都可以声明自己模块遵守的 License。比如遵守 GPL 的模块，就可以在自己的模块代码中添加：MODULE\_LICENSE("GPL")，或者是商业的模块可以 MODULE\_LICENSE(" proprietary")，之后就可以用另外一个模块调用本模块封装之后的函数，而另外的模块就不需要开源。只有设置了遵守 GPL 协议的模块才可以被 EXPORT\_SYMBOL\_GPL 定义导出的系统调用。

使用 cat/proc/kallsyms 命令会打印出包含了加载模块的内核当前的符号表，通过命令 more /boot/System.map 可以查看内核二进制符号列表。通过 nm vmlinux 也可以查看内核符号列表，可以显示所有在内核中的符号，模块中的符号要另行查看。通过 nm module\_name 可以查看模块的符号列表，但是得到的是相对地址，只有加载后才会分配绝对地址，如图 2-1 所示。

```

root@ubuntu:~# cat /proc/kallsyms |more
0000000000000000 A irq_stack_union
0000000000000000 A __per_cpu_start
0000000000000400 A exception_stacks
0000000000000900 A gdt_page
000000000000a000 A espfix_waddr
000000000000a008 A espfix_stack
000000000000a020 A cpu_llc_id
000000000000a040 A cpu_llc_shared_map
000000000000a080 A cpu_core_map
000000000000a0c0 A cpu_sibling_map
000000000000a100 A cpu_info
000000000000a1e0 A cpu_number
000000000000a1e8 A this_cpu_off

```

图 2-1

### 3. 模块参数

模块可以在编程的时候指定其接受的参数，这个参数是给用户用的。模块加载之后，用户空间通过“`echo -n ${value} > /sys/module/${modulename}/parameters/${parm}`”就可以修改模块参数。

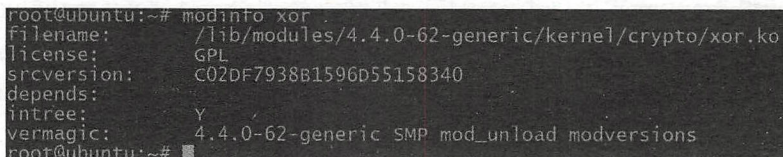
### 4. 模块的加载和卸载

模块机制存在的意义就是动态加载和卸载，原则上内核模块在被使用的过程中不可以被卸载，但可以强制卸载，或者是找到所有使用单位按照顺序关闭或卸载，使模块的引用计数变为 0。而加载的时候必须保证模块与运行中的内核相容。`insmod`、`rmmod` 分别是约定的用户端加载和卸载模块的命令，但是也可以调用内核 API 来写其他名字的命令完成同样的操作。

### 5. 模块签名

由于模块可以是外部代码，内核的版本又有很多个，所以内核必须确保该模块是使用当前内核代码编译出来的，否则执行时会出现错误。每个模块在编译时都会从内核目录中获得版本号写入编译的模块，运行中的内核在插入新的模块时会检测签名是否一致，若不一致就不会加载。

模块签名有两层含义，一层是版本号；另一层是哈希签名。使用 `modinfo` 就都可以发现，如图 2-2 所示。



```
root@ubuntu:~# modinfo xor
filename:       /lib/modules/4.4.0-62-generic/kernel/crypto/xor.ko
license:       GPL
srcversion:     C02DF7938B1596D55158340
depends:
intree:        Y
vermagic:      4.4.0-62-generic SMP mod_unload modversions
root@ubuntu:~#
```

图 2-2

内核所关心的是在图 2-2 中显示的 `vermagic`，这里没有对模块进行签名。如果签了名，则会在 `modinfo` 中多出 `signer`、`sig_key`、`sig_hashalgo` 这 3 个域。有的内核如果在编译的时候选择了 `CONFIG_MODULE_SIG_FORCE` 宏，那么没有签名的模块都是拒绝加载的。

## 2.2.2 模块编程可以使用的内核组件

### 1. workqueue

Linux 下的工作队列是一种将工作推后执行的方式，其可以被睡眠、调度，与内核线程表现基本一致，但使用起来又比直接使用内核线程简单，一般用来处理任务内容比较动态的任务链。每个 workqueue 都可以添加多个 work（使用 queue\_work 函数）。

系统有默认的 workqueue 内核线程，然而用户可以自己定义 workqueue，每一个 workqueue 都有对应的内核线程，但不是每一个 workqueue 都活跃到和其他 workqueue 所需要的资源一样的程度。再考虑到 workqueue 在使用过程中的一些其他问题，内核开发者实现了一个内核线程池，将后台承载的线程动态地绑定到 workqueue 上，这样就不需要每一个 workqueue 都创建自己的内核线程了，这个机制叫作 cmwq（Concurrency Managed Workqueue）。示例如下：

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>
static struct workqueue_struct *queue=NULL;
static struct work_struct work;
static void work_handler(struct work_struct *data)
{
    printk(KERN_ALERT"work handler\n");
}
static int __initqueue_init(void)
{
    queue=create_singlethread_workqueue("hello world");
    INIT_WORK(&work,work_handler);
    schedule_work(&work);
    return 0;
}
static void __exit queue_exit(void)
{
    destroy_workqueue(queue);
}
```



```
MODULE_LICENSE("GPL");
module_init(queue_init);
module_exit(queue_exit);
```

以上是一个最简单的 workqueue 内核模块，除了可以使用 create\_singlethread\_workqueue 创建 workqueue 之外，还可以使用 create\_workqueue 创建，此内核会为每一个 CPU 创建一个线程来执行 workqueue。这两个是常用的“老式”的自己控制线程数的接口，cmwq 使用 alloc\_workqueue 来使用线程池创建 workqueue。

## 2. 中断系统和 tasklet

Linux 中的中断分为 3 个层次。最低的层次是在源代码 arch 目录下与各个平台相关的代码，一般位于平台代码下面的 irq.c 文件中，该部分代码直接与硬件相关，最后都要调用 do\_IRQ(\_\_do\_IRQ) 进行执行。

do\_IRQ 就是中断系统的中层，其根据下层传来的中断号找到对应的中断处理函数，处理多 CPU 访问和中断重入问题，然后调用真实的中断处理函数，也就是中断的上层。但是这里内核做了区别，如果内核判断中断发生了嵌套（同时发生的中断很多）或者有其他的高时间成本的需求，则将中断处理函数以内核线程的形式（软中断）运行，否则直接运行。中断的最上层则与各个中断的具体功能相关。

tasklet 一般专用于中断，因为中断不能阻塞，所以耗时较长的操作都交给 tasklet 在中断上下文之外调度执行，基于软中断实现。软中断被内核直接使用，但是如果用户模块想要直接使用则会非常难，因为需要考虑在不同 CPU 上的调度问题，所以软中断是锁密集型的机制。内核线程 ksoftirqd 专门用来调度软中断，而模块开发的时候希望使用这种软中断的延时执行机制，就可以调用内核封装好的 tasklet 装置。

中断系统是一个非常复杂的子系统，除非深度的内核开发者，例如比较细节的多 CPU 中断、中断亲和度、中断域等概念都是不太容易接触到的。其中中断亲和度常被运维人员用于锁定应用性能。

如图 2-3 所示是可能的中断号，在每一个中断号下面的文件都可以进行中断亲和度的绑定，将特定的进程绑定到特定的中断号上，这样进程不容易被抢占。

```
root@ubuntu:~# cat /proc/irq/
0/          34/          45/          56/
1/          35/          46/          57/
10/         25/          36/          58/
11/         26/          37/          59/
12/         27/          38/          6/
13/         28/          39/          5/
14/         29/          4/           50/
15/         3/          40/          51/
16/         30/          41/          52/
17/         31/          42/          53/
18/         32/          43/          54/
19/         33/          44/          55/
                    default_smp_affinity
```

图 2-3

## 2.3 特殊硬件框架

Linux 支持很多硬件框架，下面进行简单的概括。

RAPID I/O 是一种物理连接方式，也有对应的软件驱动。用于芯片到芯片、板到板的连接，可作为嵌入式系统的背板连接，在非行业专用系统中很少见。

Linux 硬件系统中可以包含 FPGA 芯片，由于 FPGA 可被硬件随意编程为实现特定功能的组件，其实现的功能是纯硬件的，又要被 Linux 操作系统所能利用，所以就需要一个内核中存在的基础设施来驱动 FPGA 以导出给用户使用。这个驱动组件就是 XillyBus。用户必须在 FPGA 中将 XillyBus 模块的 IP 核放入 FPGA 硬件，内核中会运行一个 XillyBus 的数据转发模块，导出到用户空间供用户使用。由于这是一个通用性的组件，所以无法确切地指导数据流动的特点，因此其数据采用 FIFO 缓存。在用户空间的设备为：`/dev/xillybus_*`。

```
$ cat mydata> /dev/xillybus_thisfifo
$ cat /dev/xillybus_thatfifo>hisdata
```

如此就可以读写其中的数据了。

一个板子上可能有多个 CPU 同时在运行多个操作系统，这些操作系统可能可以共享物理内存，也可能是分割的。这种架构在高端的嵌入式板子上很常见，这些板子上共享内存的操作系统之间也需要通信，但是如果采用传统的 socket 通信，那么代价太大。内核需要一种可以让两个 CPU 直接访问的缓存作为通信空间，从而创建一个通信协议，这种机制叫作 `rpmsg`，在很多上游厂商的高端 BSP（板级支持包）中都有类似的实现。一般的大型嵌入式系统都会自己实现一个 CPU 之间通信的协议，但大致上都是使用内存，将一块内存划分为一块块信道。这种板子还有谁先启动的问题，一般的做法是先启动一个操作系统（该操作系统叫作主操作系统），另外一个操作系统由先启动的操作系统来启动。主操作系统不但可以控制启动，还可以控制关闭重启、远程过程调用等。这个框架功能就叫作 `remoteproc`。

还有“玩”嵌入式常遇到的 PWN 用于控制电机、LED 等通用接口，类似于软件层次的 GPIO。还有 PIN Controller，很多硬件设备都有很多可以配置的引脚，通常是通过一系列寄存器对其进行配置和管理，而这些可用配置的种类又大致相同的，因此就产生了抽象化的需求。这些引脚的配置空间抽象化为 PINController 注册到内核的 PINControll 子系统中统一管理。还有例如“玩树莓派”这种板子经常用到 GPIO

接口，也在/dev目录下，但是一般会通过上层封装的库进行访问使用。

时钟是特殊硬件的一大类，这些硬件有物理存在的，也有虚拟的，有服务于网络时间同步的，也有服务于单机高精度时钟的。常见的有 IEEE1588、NTP、SNTP、PTP、PPS、Watchdog、RTC、PIT、TSC、HPET、ACPI PMT 这几种。其中 IEEE1588、NTP、SNTP、PTP 是服务于网络时间同步的机制或者设备，PPS、Watchdog、RTC 在/dev目录下都有对应的时钟设备，可以给用户直接使用。PIT 作为单机时钟信号提供者现在基本被淘汰，TSC、HPET、ACPI PM-Timer 都是用于对 PIT 的替代产品。

IEEE1588 定义了一种新的时钟同步方式。该方式的出现是因为局域网内的高精度同步没有很好的产品。NTP 和 SNTP 这两种网络时钟同步算法的精度不能满足需求。PTP 借鉴自 NTP，主要思想是通过一个同步信号使周期性的设备与全网络中的设备同步校准。在一个网络中只有一个主时钟，用来产生最高精度的信号，其他的都为边界时钟，用来接收主时钟的同步信息来调整自己。由于 PTP 大部分用于局域网，而民间的局域网应用少有如此高的时间同步需求，所以 PTP 仍不被常用。

PPS 是 Linux 内核抽象出的一种设备，位于/dev/pps\*，PPS 设备每一秒钟会发送一个脉冲。系统可以使用这种设备做到时钟同步，或完成其他定时操作。而 Watchdog 则要求用户必须在一定的时间间隔内向这个设备写入数据，否则该设备就会认为程序死亡，主动重启系统。Watchdog 比较适用于程序与系统一体的嵌入式系统，但互联网服务器很少使用 Watchdog，一般在出现紧急问题时都会人工尝试恢复。Watchdog 的设备文件是/dev/watchdog。

RTC 是在启动时经常使用的时间机制。PC 电脑都有一个在离线状态下还可以运行的时钟。这个时钟在运行期可以看作是准确的、实时的，但是由于硬件原因，长期运行产生偏差也是不可避免的。Linux 内核在启动的时候会去查询这个值，并用来维护自己的时间信息，启动后大部分 Linux 都会使用网络时间来重新确定本机的时间，还会向 RTC (Real-Time Clock) 硬件写入，用来校准。由于 RTC 硬件的时间是存储在寄存器中的，一般存储的都是自某一个时间（1900 年或 1970 年）以来的秒数，而寄存器的大小是有限的，所以不同系统对这个算法的做法不一样。例如 U-Boot 读取一个值加上“1900 年”就是现在的时间，但是 Linux 除此之外会判断如果小于 1969 年，会加上 100 年得到现在的时间。由于算法不一样，所以在 Bootloader 中与在 Linux 中看到的时钟时间不一样。

RTC 是硬件也是一个软件子系统。RTC 软件子系统的存在，使得不同的硬件时钟对于系统软件透明，省去了编程的麻烦。RTC 与其他模块类似，也定义了设备，



可以供用户在/dev 目录下访问，叫作 rtc 或 rtcn (n 为整数，一个硬件系统可能会有多个 RTC 时钟，但大部分 PC 只有一个 RTC 时钟)。大多数的 RTC 带有中断功能，常见的 x86 系统中的 8 号中断就是时钟中断，内核可以使用该中断功能周期性地执行自己的任务。用户端也可以通过 RTC 设备使用这个中断机制。打开这个设备文件后，使用 ioctl 设置频率后，周期性地去读取这个设备值，就能测量时间。因为设置频率就是设置了该时钟触发 8 号中断的频率，读取设备值得到的就是自上次读操作至今的中断数目。因此，每读一次就可以得到当前过去的时间。这个时间的粒度和准确度可以设置不同的频率和读取频率来控制。RTC 用户端设备文件一次只允许一个用户单独打开。如图 2-4 所示。

```
root@ubuntu:/proc/1839# cat /sys/class/rtc/rtc0/date
2017-02-22
root@ubuntu:/proc/1839#
```

图 2-4

你可以在 sys 文件系统中查看 RTC 硬件的当前详细信息。我们平时常用的时间命令 date、hwclock 也是读取和设置 RTC 的。

虽然查询时间时可以查看 RTC，但是 RTC 作为定时器频率过低，很难满足现代应用的定时需求。Linux 内核中有一个通用的时钟抽象层，叫作 timerkeeper，timerkeeper 有一个 clocksource 的时钟源抽象封装。我们可以通过查看 sys 文件系统确定当前可用的和正在使用的时钟源。如图 2-5 所示。

```
root@ubuntu:~# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
root@ubuntu:~# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
tsc
```

图 2-5

由图 2-5 所示可以看到 RTC 并没有在可用的时钟源里。acpi\_pm 的精度也相对一般，虽然 hpet 的精度很高，但是访问 hpet 的成本也相对较高。

## 2.4 特殊软件机制

### 1. UIO

VFIO 用来取代 UIO 的框架，允许用户端直接访问设备细节，也就是说让用户

端设备驱动成为可能。其主要的工作成果是用户端可以配置 IOMMU，从而让用户端也可以在编程时使用 DMA。不过由于是新事物，其目前还仅支持 PCI 设备的驱动访问（vfio-pci 模块），另外对 CPU 的 IOMMU 配置，也只实现了 x86 和 PowerPC 两种。

用户端的设备文件是 `/dev/vfio/N`，用户可以使用这个设备文件实现完全的设备驱动程序，目前的主要用途是虚拟机时的设备驱动透明访问。

UIO 是一个在用户端实现内核驱动的机制。其在内核中有一个 UIO 模块，目前该模块只支持字符设备。用户可以添加多个 UIO 设备（用户端的设备驱动），每个设备在 `/dev/uioX`，X 为数字，第一个为 0，依次类推。我们知道设备都是靠中断来响应的，响应 UIO 设备中断的方法是读取 `/dev/uioX` 文件，没有中断的时候读取会阻塞，来中断的时候会读取到整数值，代表已经发生的中断的次数。

有的设备有多个中断，有的没有中断。针对这些情况 UIO 模块也实现了对应的机制，但是称不上完美。对于多个中断的情况，对 `/dev/uioX` 进行 `write()` 系统调用可以打开或者关闭内核的中断处理，以便驱动可以手动处理中断。在没有中断的情况下，UIO 模块提供了一个定时器接口，通过设置这个接口可以人工地让这个设备定时产生中断。

UIO 用户定义的驱动类似于内核驱动，一般会有一些需要通过 `sys` 文件系统访问的全局变量。UIO 模块不支持调用 `sysctl` 更改这些值，但是可以在 `sys` 文件系统找到这些对应的文件，从而进行修改：`/sys/class/uio/uioX`。如图 2-6 所示。

```
root@ubuntu:/sys/class/uio/uio1# ls
dev  device  event  name  subsystem  uevent  version
```

图 2-6

每个 UIO 设备都有 `name`、`version`、`event` 这 3 个定义属性，还有一个 `maps` 文件夹（有内存映射的时候才存在），其他为 `sys` 文件系统自带的 `uevent` 模型。`name` 表示这个 UIO 设备的名字；`version` 用于表示当前的 UIO 内核模块的版本；`event` 与 `read()` 设备获得的值一样，是当前已经发生过中断的次数。`maps` 文件夹服务于硬件的数据处理。大部分硬件都需要操作内存，UIO 用户驱动如果要映射设备内存到用户端操作，需要使用 `mmap` 系统调用，每使用一次系统调用会在 `maps` 目录下生成一个目录 `map[digital]`，里面有 4 个固定的文件用来描述映射的内存的信息，如图 2-7 所示。

```
root@ubuntu:/sys/class/uio/uio0/maps/map0# ls
addr name offset size
```

图 2-7

其中 `addr` 是映射的基地址；`offset` 是偏移量；`name` 是映射时给这段映射起的名字；`size` 是映射的内存的大小。如果不能映射内存，还可以通过 x86 的端口操作系统调用 `ioperm()`、`iopl()`、`inb()`、`outb()` 等对某个硬件端口进行读写。在这种情况下，UIO 模块还添加了 `/sys/class/uio/uioX/portio/` 目录。

由于 UIO 在用户空间写驱动的便利性，所以对 FPGA 提供了很好的支持，甚至 DPDK 这种将内核数据包导出到用户空间的机制中的 kni 网卡驱动也使用 UIO。Open Source Automation Development Lab 等使用机器人编程的组织也喜欢 UIO。

## 2. VFIO

VFIO 则是软件对硬件设备内存暴露在用户空间的支持，是对 UIO 的升级。DMA 内存直接被映射到用户进程空间，使用这个驱动需要将设备与操作系统原来的驱动解绑。目前仅实现了支持 `vfio-pci` 模块和 PCI 设备的映射。这对于在虚拟机和用户空间设备驱动有重要意义。访问设备内存的机制被单独称为 IOMMU。IOMMU 本来是为虚拟化而设计的，使用场景是如果驱动在用户态（比如虚拟机），没有高效的使用设备 IO 内存的方法，内核要在用户内存空间到设备内存空间做额外的转换，IOMMU 可以直接将设备的内存空间映射到用户进程空间。用户可以通过直接操作硬件的内存空间来操作硬件。

## 3. SysRq

SysRq 机制很容易被人忽视，使用此机制解决特殊问题是非常有效的。SysRq 类似 Windows 的“Ctrl+Alt+Del”组合键的效果，只要系统不是处于完全被锁死的状态，就会优先响应这个命令。在 Linux 中这个功能本身是可以打开或关闭配置的，使用 `echo "1" > /proc/sys/kernel/sysrq` 打开 SysRq 机制。在 Linux 中调用该系列命令的方式是“SysRq 键+命令”。SysRq 在大部分键盘上一般是 Print Screen 按键的副功能，需要使用 Alt 键调用。与 Windows 操作系统不同的是，Windows 操作系统一定是在按键后跳出图形界面，而 Linux 允许直接使用按键命令执行特定的操作，下面列出常用的一些功能。

- SysRq+b: 立即重启系统。



- SysRq+c: 立即产生一个系统级的 crash dump。
- SysRq+d: 显示当前使用中的所有锁。
- SysRq+e: 发送 SIGTERM 给出 init 之外的全部进程。
- SysRq+f: 手动调用 oom killer 杀死当前 OOM 打分最高的进程。
- SysRq+g: 被 kgdb 使用。
- SysRq+h: 显示 SysRq 的使用帮助。
- SysRq+i: 发送 SIGKILL 信号给除了 init 外的所有进程。
- SysRq+j: 相当于调用 `ioctl(fd, FIFTHAW, 0)`; 让一个被冰冻的文件系统 (`ioctl(fd, FIFREEZE, 0)`;) 解冻。
- SysRq+k: 杀掉当前虚拟终端上开启的所有进程。
- SysRq+l: 显示所有 CPU 的调用栈。
- SysRq+m: 导出当前的内存信息。
- SysRq+n: 使采用实时调度算法的进程可以设置 nice 值。
- SysRq+o: 关机。
- SysRq+p: 打印当前寄存器。
- SysRq+q: 打印当前各个时间装置 (如图 2-8 所示)。

```

[1226523.088333] Tick Device: mode: 1
[1226523.088670] Broadcast device
[1226523.088993] Clock Event Device: hpet
[1226523.089320] max_delta_ns: 149983013277
[1226523.089645] min_delta_ns: 13410
[1226523.089940] mult: 61496111
[1226523.090217] shift: 32
[1226523.090475] mode: 1
[1226523.090771] next_event: 9223372036854775807 nsecs
[1226523.091036] set_next_event: hpet_legacy_next_event
[1226523.091320] shutdown: hpet_legacy_shutdown
[1226523.091628] periodic: hpet_legacy_set_periodic
[1226523.091904] oneshot: hpet_legacy_set_oneshot
[1226523.092175] resume: hpet_legacy_resume
[1226523.092439] event_handler: tick_handle_oneshot_broadcast
[1226523.092721] retries: 0
[1226523.092983]
[1226523.093239] tick_broadcast_mask: 00000000,00000000,00000000,00000000
[1226523.093528] tick_broadcast_oneshot_mask: 00000000,00000000,00000000,00000000
[1226523.094171]
[1226523.094456] Tick Device: mode: 1
[1226523.094783] Per CPU device: 0
[1226523.095068] Clock Event Device: lapic
[1226523.095431] max_delta_ns: 1377516535820
[1226523.095732] min_delta_ns: 1000
[1226523.096081] mult: 13391305
[1226523.096368] shift: 27
[1226523.096724] mode: 3
[1226523.097006] next_event: 1226512319129364 nsecs
[1226523.097310] set_next_event: lapic_next_deadline
[1226523.097631] shutdown: lapic_timer_shutdown
[1226523.097954] periodic: lapic_timer_set_periodic
[1226523.098247] oneshot: lapic_timer_set_oneshot
[1226523.098609] event_handler: hrtimer_interrupt
[1226523.098903] retries: 87193
[1226523.099473]

```

图 2-8

- SysRq+r: 将键盘输入设置为 XLATE 模式。Raw 模式是平时使用的, 在这种模式下, 键盘返回。输入键位编码给计算机, 而在 XLATE 模式下, 键盘直接输出 ASCII 字符给计算机。
- SysRq+s: 同步所有已经加载的文件系统。
- SysRq+t: 打印当前所有的任务和详细信息。
- SysRq+u: 把所有的文件系统都 unmount, 然后用只读的权限重新 mount。
- SysRq+w: 打印所有处于阻塞状态的进程。
- SysRq+y: 打印所有寄存器。
- SysRq+z: 导出 ftrace buffer。
- SysRq+0~9: 设置内核的 log 级别。

这些命令视内核的配置而部分有效, 这在运维工作中也是相当有用的。我们可能会遇到 Shell 登录上去只能输入命令, 但是大部分命令不会被执行的情况, 笔者在 Ubuntu 12 上调试 DPDK 的 kni 模块时就频繁地遇到此问题。这个时候只能重启机器, 而重启的方法要么是让运维人员重启, 要么是自己去实验室重启。但是使用 SysRq 就能解决这个问题, 因为它提供了不需要执行外部命令的重启方式。

#### 4. 其他机制

有一些逻辑虽然是串行的, 但是可以分成不同的部分, 每一部分可以分别执行。PADATA 就是这样就可以在多个 CPU 上同时执行不同的逻辑, 但是又能保证逻辑的顺序机制, 这个机制最早是为 IPSec 开发的。

namespace 机制在出现很长时间后都没有被重视, 直至虚拟化成为重要需求。现在 pid、net、ipc、mount、user 等都逐渐支持 namespace, 并且逐步诞生了 cgroup, 从而使得 docker 成为了可能。

内核中很多地方都有使用引用计数的需求, 涉及资源回收和资源竞争, 或者是访问统计等。这种需求一般是使用一个整数, 自己编写的时候控制其增加或减少。而控制的时候又要考虑并发冲突等很多情况, 通常要自己封装函数。Linux 就实现了一种通用的数据结构和相关函数调用, 使用者直接使用接口即可。示例如下:

```
struct my_data
{
    .
    .
    .
}
```

```
struct krefrefcount;
```

```
};
```

我们在 Shell 中敲入的命令必须是 Shell 解析器内置的，或者是位于 PATH 环境变量设置的路径中可以找到的 ELF 格式的可执行文件，例如 dd 命令一般位于 /bin/dd，当在 Shell 中输入 dd 的时候，Shell 解析器就会去 /bin 文件夹下寻找 dd 二进制，而 /bin 一般是在 PATH 环境变量中设置的。Linux 不仅可以支持 ELF 格式的二进制文件，其他格式的程序执行也可以通过其他的解析器来支持。例如通过 Python 解释器可以执行 Python 的程序，通过 Perl 解析器可以执行 Perl 程序，通过 PHP 解析器可以执行 PHP 程序，甚至 Shell 脚本本身也是一种程序，可以用 Shell 解析器（例如 bash 程序）解析执行。

内核提供了一种方法允许将例如 Java 的程序与 ELF 二进制一致看待。用户只需要在 Shell 中敲入 Java 程序名（或者 Python 程序名），只要该程序在 PATH 下就可以像 ELF 格式可执行程序一样被执行。做到这样的方式是使用 binfmt\_misc 机制，该机制通过 proc 文件系统操作，若要使用首先要先 mount 上去：mount binfmt\_misc -t binfmt\_misc /proc/sys/fs/binfmt\_misc。然后向 /proc/sys/fs/binfmt\_misc/register 中写入规定格式的字符串即可，如下所示。

```
:name:type:offset:magic:mask:interpreter:flags
```

其中冒号是必须的，不同需求替代不同的字符串。例如对 Wine 程序的支持使用：:Wine:M::MZ::usr/bin/wine:写入 register 文件。

Shell 解析器本身也提供了很好的调用其他解析器的能力，而不需要使用 binfmt\_misc。例如 Bash 解析器就支持在文件的第一行使用 #!/bin/python 作为第一行，从而就会调用 /bin/python 来作为解析器解析后文的脚本程序。但是这种方式对于不同的二进制格式就无能为力，一般用于支持不同解析器的脚本。这种首行设置解析器的功能已经成为 Linux 的所有 Shell 约定俗成的规定了。



## 内核数据结构

内核数据结构并不与用户端的数据结构有什么理论上的不同，但是内核数据结构是为内核常见的目的而专门设计的数据结构，通常比用户端的通用数据结构更加注重性能和业务场景。本章简单介绍内核数据结构，FIFO 文件的使用属于用户空间的高级进程通信话题，本章详细介绍。

### 3.1 链表与哈希表

链表之所以被称作链表，是因为它的弹性不动属性。它可以把离散时间到达的数据结构串起来，使其可以更容易地被索引，并且不需要移动之前的内容。内核哈希表是由链表群组成的，其每一个哈希桶（每一个哈希的值位置）都是一个链表。高性能的哈希表有很多种实现方法，例如 DPDK 里就有几种现成的，但是后来被 DPDK 所采用的用来处理哈希冲突的 Cuckoo Hash 算法，却只能保证哈希表使用率较低的情况下的高性能，换句话说相对不可控的，这显然与 Linux 的需求不相符。内核中采用的是开链的哈希表，但是 Linux 内核中全部使用双向链表，而这个链表在应用到哈希表做开链哈希桶时要有针对性的优化。

### 3.1.1 双向链表

双向链表很容易做成一个环，将起始的 `prev` 域设为最后一个数据，将最后一个数据的 `next` 设为起始数据。一般链表的设计思路是以链表节点为主体，一个双向链表节点的数据域一般包括 `prev`、`next` 与 `data`。然而这样设计的一个明显缺点就是当一部分代码要处理链表的一部分数据时，其拿到的永远是链表的节点，要操作里面的数据代码需要通过节点索引到具体的数据位置。如果这种代码块很多，那么每次都要进行索引操作，这对于代码的“整洁性”就是败笔。

Linux 采用了一个取巧的做法（`include/linux/list.h`），即双向链表的节点不包含实际的数据域，而是数据结构中包含链表的节点。理论上链表功能是数据结构功能的一部分。当一段代码拿到一个数据结构的实例后，就可以通过访问其中的链表域来确定其链表情况，使用链表定位到某一个实例后就可以使用 `container_of` 宏定位到这个结构体本身。

`container_of` 宏的作用是已知结构体某个域的指针，求出结构体实例的指针。原理是根据结构体定义得出已知指针的偏移，用已知指针减去这个偏移就是结构体的指针。代码如下。

```
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *) ( (char *)__mptr - offsetof(type,member) );})
```

因此这个 `container_of` 并不只用于链表节点到真实数据结构体的映射，其他域都可以使用。代码如下。

```
struct list_head {
    struct list_head *next, *prev;
};
```

这个双向链表的核心是一个头部，其与用户端链表的显著不同是并没有数据域。

### 3.1.2 hlist

双向链表的缺点是如果有很多特别短的链表时（很可能只有一个节点），双向链表的 `next` 和 `prev` 的头部就非常占用空间。典型的是哈希表，我们知道哈希表使用哈

希函数计算得到一个地址，然后直接访问该地址的机制实现快速访问，但是哈希算法不可避免地会有哈希冲突（多个输入产生了同一个地址输出），此时解决哈希冲突的方法就是使用哈希桶，一般在同一个计算地址的位置实现一个链表，该链表链出所有哈希结果为本地址的值。

通常情况下，哈希表大部分的域都是空白的，而哈希表所需要的大小却要提前分配，只有每个哈希桶的链表才可以动态分配。一个双向链表的头部有两个指针的大小，如果这两个指针全部放入哈希表要提前分配空间，就会比单链表消耗多一倍的内存空间。所以内核专门设计了 hlist，拥有只有一个指针大小的头部的双向链表。如图 3-1 所示。

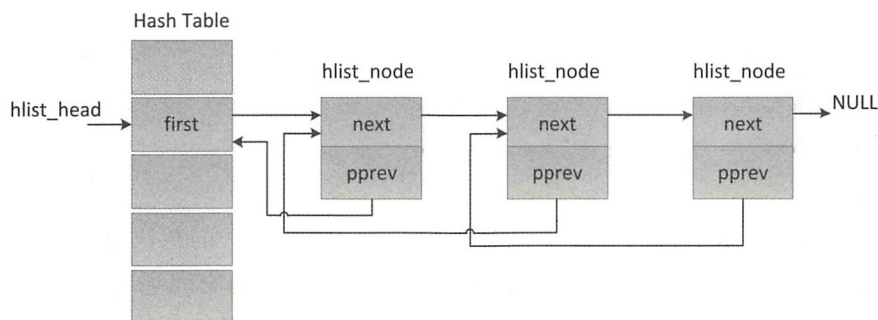


图 3-1

哈希链表本质上也是双向链表，但是组织方式与双向链表并不完全一样。代码如下。

```
//哈希桶的头结点
struct hlist_head {
    struct hlist_node *first; //指向每一个hash桶的第一个结点的指针
};

//哈希桶的普通结点
struct hlist_node {
    struct hlist_node *next; //指向下一个结点的指针
    struct hlist_node **pprev; //指向上一个结点的 next 指针的地址
};
```

Linux 双向链表是环形的，而 hlist 有特定的头部结构。其头部只有一个 first 域，用来指向第一个节点（一个头部只有一个整数大小）。而第一个节点和后续的节点的



结构都是一样的，只是第一个节点的 `next` 和 `pprev` 都是指向头部的 `first` 域。后续节点的 `next` 指向的是下个节点的 `next` 域（就是下一个节点的地址），而 `pprev` 指向的是上一个节点的 `next` 域。在整个链表的设计中，与双向链表的不同有两处，即头部和 `pprev`。而 `pprev` 本质上又和双向链表的 `prev` 是一样的，都是指向上一个链表数据结构的地址。代码如下。

```
static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    WRITE_ONCE(h->first, n);
    n->pprev = &h->first;
}
```

这么做其实是为了适应头部设计而引入的带“副作用”的解决方法。我们前面讨论过之所以要设计头部为一个整数大小的原因（哈希桶的空间利用率），而这个头部的引入，导致了第一个节点和后面节点的不同。为此所有相关的操作（添加、删除、遍历等）理论上都得特别照顾第一个节点（多一个特殊情况判断）。如果能从数据结构设计上将这种特殊情况去掉无疑是最好的，`hlist` 的设计做到了这一点。其 `pprev`（对应于双向链表的 `prev`）并不是指向上一个节点的数据结构，而是指向上一个节点的 `next` 域。这对于头部节点来说，`hlist_head` 结构体的 `first` 域既是指向下一个节点的指针，也是下一个节点的上一个节点位置（通过 `pprev` 指向），如此就能让第一个节点和后续的节点对所有操作展现出几乎一样的接口。

### 3.1.3 ScatterList

该数据结构存在的原因是系统运行时会产生内存碎片，为了利用内存碎片传送大数据，在 DMA 支持分离的多块内存同时的传输下，对应产生的软件结构，其表示的是多块分离的块内存。

`scatterlist` 的主要结构体如下。

```

struct scatterlist {
    unsignedlong page_link;    //指向下个 scatterlist 节点的地址
    unsignedint offset;        //偏移
    unsignedint length;        //长度
};

```

这里的数据结构的命名与设计不容易理解,但是其组织很高效,核心在 `page_link` 域。`page_link` 域指向数据的存储内存页, `offset` 表示其指向的内存存在本内存页内的偏移, `length` 表示其指向的内存空间的大小(可以大于一个页)。`scatterlist` 的数组称为 `scatterlist table`。

Linux 允许两个 `scatterlist` 被连接起来, 构成一个更大的 `scatterlist` (如果多次调用, 就可以连接起很多个 `scatterlist`)。级联可以使用该结构体的 `page_link` 域。`page_link` 不论是指向一个 `page` 页地址、一个 `scatterlist` 结构体, 还是指向存储的页地址, 其最后两位因为对齐原因理论上都是 0 (可用的不止两位, 但只用了两位)。所以就可以使用最后两位来区别前面的地址到底是页地址还是 `scatterlist` 地址: 01 表示地址指向的是 `scatterlist` (级联情况); 10 表示后面没有级联的节点了; 00 表示 `page_link` 指向的是存储的页地址。

当一个 `scatterlist` 结构体用作级联时, 该结构体内的其他域都为 0, 表示该节点不指向任何可用的存储空间, 而只是用来连接下一个 `scatterlist table` 的桥梁。代码如下。

```

struct sg_table {
    structscatterlist *sgl;    /* the list */
    unsignedint nents;         /* number of mappedentries */
    unsignedint orig_nents;    /* original size of list*/
};

```

`sg_table` 是对 `scatter_list` 的总体概括, 内含了 `scatterlist`、一个 `map` 之前的块数目 `orig_nents`, 以及一个 `map` 之后的块数目 `nents`。这里的 `map` 的意义是各个分离的内存有可能恰好相邻, 并可以合并。

`scatterlist table` 可以被拼接 (chain), 如果不同的 `scatterlist` 所指向的内存是相邻的还可以被合并, 所以其遍历格外复杂。

### 3.1.4 llist

`llist` 的全称是 Lock-less NULL terminated single linked list, 意思是不需要加锁的

list。在生产者-消费者模型下，如果有多个生产者和多个消费者，生产者意味着链表添加操作，消费者意味着链表删除操作。但多个生产者或者多个消费者一起操作时就需要加锁，但加锁是高耗费的操作，内核现在流行无锁操作，为这个需求诞生的就是 llist。

Linux 中断有上半部分和下半部分。上半部分会关闭所有中断，使系统失去响应，为了减少这个时间，上半部分不能休眠，复杂的操作也得放到下半部分执行。也就是意味着上半部分的代码不能使用加锁操作，而中断重入是很容易发生的，这就又诞生了加锁的需求。提高上半部分使用 list 数据结构能力的最好的方式就是使用无锁 llist。代码如下。

```
struct llist_head {
    struct llist_node *first;
};
struct llist_node {
    struct llist_node *next;
};
```

内核实现的方法是使用 cmpxchg 宏。这本来是一个 Intel 平台的汇编指令，Linux 喜欢这个指令，但其他平台不一定有，所以就封装了一个宏，在其他平台重新实现，而 Intel 平台直接调用 cmpxchg 指令即可。这个指令是原子的，有 3 个参数，对比第一个和第二个参数，如果相等则写入第三个参数到第一个参数指向的地址。如果不相等则返回第三个参数。本意是将第三个参数写入第一个参数指向的地址，但是加上对比后在写入之前第一个参数指向的内存就不会被其他任何指令修改，这就确保了在写入操作之前和之后的一致性。具体到 llist 就是在添加链表节点到头部时，需要确保在修改 head->first 指针时，没有其他的并行操作也在修改，因为同时修改会造成并发的操作，产生无法预知的后果。所以 cmpxchg 先读出 head->next，然后 head->first 作为第一个参数，刚读出的 head->first 作为第二个参数，要添加的新节点作为第三个参数。代码如下。

```
bool llist_add_batch(struct llist_node *new_first, struct llist_node
*new_last, struct llist_head *head)
{
    struct llist_node *first;
    do {
        new_last->next = first = ACCESS_ONCE(head->first);
```



```
    } while (cmpxchg(&head->first, first, new_first) != first);  
    return !first;  
}
```

## 3.2 其他数据结构

### 3.2.1 树

Linux 内核中实现了一个通用的 B+树(btree.h), B+树大部分被用于文件系统中, 一些文件系统也实现了自己的 B+树。内核中还有 radix tree 用于将指针与 long 整数键值关联(例如 IDR 机制)。Linux 基树与 trie 树非常类似, 只是一个用来查询定位整数, 一个用来定位字符串。

IDR (Integer ID Management) 机制在 Linux 内核中指的是整数 ID 管理机制, 即将一个整数 ID 号和一个指针关联在一起的机制。IDR 机制最早在 2003 年 2 月加入内核, 当时作为 POSIX 定时器的一个补丁。现在内核中很多地方都可以找到它的身影。

IDR 机制适用于那些需要把某个整数和特定指针关联在一起的地方。例如在 IIC 总线中, 每个设备都有自己的地址, 要想在总线上找到特定的设备, 就必须要先发送设备的地址。当适配器要访问总线上的 IIC 设备时, 首先要知道它们的 ID 号, 同时要在内核中建立一个用于描述该设备的结构体和驱动程序。将 ID 号和设备结构体结合起来, 如果使用数组进行索引, 一旦 ID 号很大, 用数组索引则会占据大量的内存空间, 这显然不可行。也可以用链表进行索引, 但是如果总线中实际存在的设备很多, 则链表的查询效率会很低。此时 IDR 机制应运而生, 该机制内部采用 radix tree 实现, 可以很方便地将整数和指针关联起来, 并且具有很高的搜索效率。

### 3.2.2 FIFO 文件

FIFO 文件是内核提供给用户空间的一个非常好用的工具, 也叫作命名管道, 其与 K\_FIFO 的内部机制并不一样, K\_FIFO 的内部是一个简单的双向队列, FIFO 文件

相当于一个跨进程的队列，并且提供了原生的阻塞和配合文件能力的工具。例如你可以用 `select` 的 `BLOCK` 模式打开这样一个 `FIFO` 文件，同时设置超时，也可以用普通的 `read` 调用读取 `FIFO` 文件内容，这样 `FIFO` 在收到数据之前将永久阻塞。如果没有 `FIFO` 文件机制，则要使用消息队列实现同样的操作，或者使用 `UNIX domain socket` 进行模拟，后者更容易一些。然而 `FIFO` 文件还有一个特性，就是它首先是一个文件，即使没有人在读取，也可以往文件里写内容，这就带来了比 `socket` 更加强大的能力。但是在严肃的应用场景下，`FIFO` 文件并不容易使用，要同时满足以下这些需求时才应该选用 `FIFO`。

- 跨进程传输数据。
- 数据的产生和数据的监听不同步，并且希望产生数据的时候读取者即使没有读取也有缓存能力。
- 数据只要被读取了就会被删除，第二次就不会被再次读取。
- 需要等待超时、永久阻塞、立即返回中的某些或者全部特性。
- 需要运维系统查看数据流。
- 目前为止没有任何办法提前判断你要读的阻塞式的 `FIFO` 内是否有数据。

在 Linux 操作系统中，命名管道 `FIFO` 文件几乎是唯一一种同时满足以上需求的通信方式，但是绝大多数情况你仍然需要的是 `Unix Domain Socket`，除非你需要随时查看或者截断通信过程中的数据流（便于调试和运维），或者是遇到了极限的性能问题（一般 `FIFO` 文件性能是最好的）。`FIFO` 文件是不容易使用的。大部分人会使用 `FIFO` 文件的非阻塞模式，如果需要阻塞等待就调用 `select`，并且设置 `timeout`，然后在事件到来的时候反复调用 `read`，直到数据读完。因为阻塞模式下 `select` 的 `timeout` 是没有效果的，文件句柄 `fd` 会一直处于可读状态。而阻塞模式只要读了就会一直阻塞，没有 `timeout`。并且在阻塞模式下，如果你调用 `read` 读取了一定的长度，就永远不知道还有没有数据没有读取完，除非继续读取，然而如果这么做就会继续被堵塞。但并不是说用非阻塞的，使用 `select` 配合就可以高枕无忧了。当你 `select` 到有内容可以读，然后读取处理，再次调用 `select`，在这期间如果没有读取到完整的数据，即使数据后续被写入到了 `FIFO` 文件中也读取不到，因为你在后续数据事件到达之后才调用了 `select`。也就是说这种模型在单个消息比较大的情况下使用是有问题的，比较简单的方法是直接使用非阻塞遍历循环的查询。但是如此就很难平衡 `FIFO` 文件缓存大小和遍历频率的设计带来的 CPU 损耗。代码如下。

```

class FifoFile{
public:
    FifoFile(string _path);
    string blockRead(uint64_t timeout);
    bool blockWrite(string _line);
    ~FifoFile();
private:
    string path;
    queue<char> charQueue;
    char* buff;
    string line;
};

string FifoFile::blockRead(uint64_t timeout){
    string result;
getFromLib:
    while(!charQueue.empty()){
        if(charQueue.front()=='\n'){
            charQueue.pop();
            result = line;
            line.clear();
            return result;
        }else{
            line += charQueue.front();
            charQueue.pop();
        }
    }
    if(line.size()>MAX_FIFO_LINE_LENGTH){
        line.clear();
        return string();
    }
}

sleep(1);
int fd=open(path.c_str(),O_RDONLY | O_NONBLOCK);
if(fd == -1){
    cerr<<"open fifo file failed:"<<path<<endl;
    return string("command:open file failed");
}
fd_set set;

```



```

    struct timeval _timeout;
    FD_ZERO(&set);
    FD_SET(fd, &set);
    _timeout.tv_sec = timeout;
    _timeout.tv_usec = 0;
    int rv = select(fd + 1, &set, NULL, NULL, &_timeout);
    if(rv == -1){
        perror("select");
        close(fd);
        return string();
    }else if(rv == 0){
        cout<<"config block read fifo timeout:"<<path<<endl;
        close(fd);
        return string();
    }
    uint64_t total_readed = 0;
    int64_t readed = 0;
    do{
        readed = read(fd, buff + total_readed, MAX_FIFO_LINE_LENGTH
- total_readed);
        if(readed == -1){
            cout<<"no data readed,total readed:"<<
total_readed<<".path"<<path<<endl;
        }else if(readed > 0 ){
            total_readed+=readed;
            if( MAX_FIFO_LINE_LENGTH - total_readed < 10){
                cerr<<"max line size reached,discard"<<endl;
                close(fd);
                return string();
            }
            sleep(1);
        }else if(readed == 0){
            cout<<"read finish this turn"<<endl;
        }
    }while(readed > 0);
    for(uint64_t i =0; i<total_readed; i++){
        charQueue.push(buff[i]);
    }

```

```

    }
    if(close(fd) != 0){
        perror("close fifo file failed");
    }
    goto getFromLib;
}

```

以上是一个可以 block read 的完整实现，这里的用处是每次读取一行数据返回。其中涉及这种读取要考虑的所有情况，可见使用其很不容易。

总体而言，读取文件分为监听式和轮询式。监听式看起来简单，但是在实际使用中有很多问题，通常需要借助线程进行辅助处理。类似 socket 接到一个连接就生成一个线程来处理的流程。而轮询式一般也是开一个线程去轮询，但是轮询一遍会睡眠，否则就会造成 CPU 占用过高。大部分情况下轮询的方式简单有效。

但是阻塞却是 FIFO 文件在交互式 shell 中非常有效的调试工具。mkfifo 命令也能建立 FIFO 文件。在阻塞模式下，当 cat 的时候 FIFO 文件里面没有数据，命令行就会阻塞在那里，但是若有一方使用了 echo 等向其写入了数据，cat 就会立即执行读取数据。FIFO 文件对于交互式的调试在使用上是十分友好的。

在 Linux 中，管道并没有用专门的数据结构，而是通过将两个 file 结构指向同一个临时的 VFS 索引节点 inode，而这个 VFS 索引节点又是指向一个物理页面而实现的。FIFO 文件被称为命名管道，而 PIPE 则是不存在实际的文件系统文件，只有程序调用 API 的匿名管道。

FIFO 和 PIPE 的最大缓存大小可以在 /proc/sys/fs/pipe-max-size 文件里修改和查看。并且当你使用 shell 的时候在 ulimit 里面还有一个 pipe size 的大小限制。即使使用命令行，在操作 FIFO 的时候也需要格外注意，比如要及时刷掉写入缓存。代码如下。

```
stdbuf -o0 cat test.txt >result.txt
```

在写入 FIFO 文件的时候最好不要使用文件的缓存功能，要一次性地完整写入。

### 3.2.3 位数组 bitmap

位数组叫作 bitmap，是以位为单位存储值的方式。很多情况下一些值只有两个

取值，而这些值又很多，就可以用位数组实现。也可以看成是 `bool` 类型的数组，但是 `bitmap` 提供了更多的相关操作，而这些操作很多都是实际使用 `bitmap` 数据结构的模块进行实际实现的。其定义非常简单：

```
#define DECLARE_BITMAP(name, bits) unsigned long  
name[BITS_TO_LONGS(bits)]
```

位数组可以用来执行排序算法，利用的就是数组中的索引值，将某个索引值设置为 1 就表示记录，然后顺序输出索引值就是排序后的结果（桶排序）。更多的时候位数组在文件系统中充当位图。几乎大部分的文件系统都有位图的概念，例如 `ext2` 中就有 `inode` 位图和数据块位图，用来表示对应序号的 `inode` 或者数据块有没有被使用。在 `raid` 系统中，例如 `raid1` 的数据一致性保障，由于系统有两份拷贝，不一致的情况就有可能发生，当保证一致时就会设置对应的位图位，后续的更新就都是增量的，只需要查看位图就知道哪些是未能保证一致的数据了。



# 4

## 第 4 章

# Linux 系统的启动

### 4.1 启动的硬件支持

#### 4.1.1 固件

##### 1. Legacy BIOS

Legacy BIOS 是传统的 BIOS，传统的 BIOS 直到 x86\_64 架构还有在使用的，原因是向下兼容。BIOS 使用 16 位的汇编代码、寄存器参数调用、静态链接、在 1MB 以下内存固定编址。在各大 BIOS 厂商的努力下，BIOS 扩展了很多功能，如 PnP BIOS、ACPI、USB 设备支持等，但根本性质没有变。

BIOS 下的设备驱动的执行方式是使用中断向量和固定大小的中断服务空间，典型的一个中断服务只有 128KB 的空间，也就是驱动大小不能超过 128KB，并且驱动也都是以 16 位的形式编写和存在的。

##### 2. EFI BIOS

现在的主板上基本都是 EFI BIOS 了，之所以不需要向下兼容的之前的 BIOS，是因为 EFI BIOS 是在一个全新的架构（Itanium）下设计出来的，没有兼容问题。

EFI (Extensible Firmware Interface, 可扩展固件接口) 的实现使用了 C 语言, 所以就有堆栈、模块化、动态库等能力, 软件工程带来的纠错性和可修改性特性缩短了开发时间, 并且不再是只有 16 位的寻址能力, 而是拥有 32 位或 64 位的寻址能力, 能够达到处理器的最大寻址, 也就是可以使用很多的内存了。其他的 BIOS 厂商也曾经试图取代 BIOS, 但是由于力量分散, 厂家太多, 无法形成统一的标准, 然而 EFI 由 Intel 主推, 已经逐步掌握整个市场。

与传统 BIOS 不同, EFI 上的设备驱动不是由汇编写成的, 也不是由 C 语言写成的, 而是 EFI 的虚拟指令集。如此就保证了驱动的 CPU 无关性, 即无论是志强还是安腾的 CPU, 同样的驱动代码都可以检测到正常的设备, 不需要重新编译。这样可复用的开发模式使得 EFI 可以在不启动操作系统前就访问网络, 甚至浏览网页。其可扩展性和可复用性带来的好处就是对于 EFI 系统的任何一次开发, 都可以长久地被复用。

但是这不是 EFI 可以实现一个完整操作系统的理由。EFI 的设计者故意使 EFI 没有实现一个操作系统的可能性。例如不支持中断, 所有的硬件状态都是通过轮询完成的, 并且其驱动代码是解释执行的。EFI 上可以实现程序, 但是所有程序都具有所有硬件的完全访问权限。所以, EFI 注定只是一个启动前的过渡, 启动完成后就会将主动权交给操作系统, 自己的大部分功能停止运行。

### 3. UEFI

EFI 是由 Intel 发起的, UEFI 是之后发展的产物, 由国际组织 Unified EFI Form 运行。这个组织除了 Intel 外还有很多其他的公司, 所以有了前面的 U 字母。想要推动一个基础协议架构, 若没有多个巨头的参与, 一般情况下是推不动的。UEFI 在 EFI 的基础上提供了图形化的操作界面, 使用鼠标操作, 我们也能在比较新的主板上看到这种操作方式。

## 4.1.2 磁盘分区管理

ROM 系统和 Bootloader 之间需要一种约定的调用规则。因为固件放在 ROM 中, 开机后是会固定执行的, 但 Bootloader 的代码却是放在磁盘 (或其他存储设备) 中的, 需要从磁盘中加载。而在磁盘中组织数据的方式是文件系统, Bootloader 或者

内核的主体应该放在文件系统中。

当固件看到一个磁盘，发现了其拥有该磁盘的驱动，也就是拥有了读写该磁盘的能力后，接下来应该获取的信息是本磁盘的分区情况。磁盘的第一个字节就是可执行代码，然而这部分代码并不位于文件系统中。在传统的 MBR 磁盘格式中，前 512 字节存储了一些启动代码和整个磁盘的分区表。但是限于大小的原因，一个 MBR 只能支持 4 个分区。随着技术的发展，人们在单个分区的开头部分又实现了级联的分区表，启动软件也都能够陆续识别。但是存放启动文件的根目录必须位于主 MBR 上的分区。主 MBR 上的分区叫作 Primary Partition（主分区），级联分区叫作逻辑分区（Logical Partition）。使用这个命令可以读取到 MBR 的二进制，根据结构定义就可以详细查看 MBR 的结构。如下所示。

```
dd if=/dev/sda of=mbr.bin bs=512 count=1
```

然而 MBR 的这种定义，包括其较小的表示容量的位数，限制了每个分区的大小和可以支持的分区总数。随着 EFI 的推出，同时新的替代方案 GPT 也出现了。传统 MBR 信息存储于 LBA 0（逻辑块），GPT 头存储于 LBA 1，接下来才是分区表本身。64 位的 Windows 操作系统使用 16,384 字节（或 32 扇区）作为 GPT 分区表，接下来的 LBA 34 是硬盘上第一个分区的开始。GPT 这样空出 LBA 0，MBR 的兼容性就得到了保障。一个磁盘设备（sdc）的 GPT 头部，如图 4-1 所示。

```
root@ubuntu:/# gdisk -l /dev/sdc
GPT fdisk (gdisk) version 1.0.1

Partition table scan:
  MBR: protective
  BSD: not present
  APM: not present
  GPT: present

Found valid GPT with protective MBR; using GPT.
Disk /dev/sdc: 83886080 sectors, 40.0 GiB
Logical sector size: 512 bytes
Disk identifier (GUID): 90631747-E536-45C7-BB95-0342C77E8FF8
Partition table holds up to 128 entries
First usable sector is 2048, last usable sector is 83886046
Partitions will be aligned on 2048-sector boundaries
Total free space is 0 sectors (0 bytes)

Number Start (sector) End (sector) Size Code Name
  1         2048      83886046 40.0 GiB 8300
```

图 4-1

一个 GPT 分区表项的前 16 个字节是分区类型 GUID。例如，EFI 系统分区（存放模块代码）的 GUID 类型是{C12A7328-F81F-11D2-BA4B-00A0C93EC93B}。接下



来的 16 字节是该分区唯一的 GUID(这个 GUID 指的是该分区本身,而之前的 GUID 指的是该分区的类型)。再接下来是分区起始和末尾的 64 位 LBA 编号,以及分区的名字和属性。

MBR 分区表又叫作 MSDOS 分区表(出于历史原因)。还有两种不太常用的分区表是 SGI 和 SUN 分区表,一般只用在对应公司的平台上。

## 4.2 Bootloader和内核二进制

### 4.2.1 Bootloader

Bootloader 是 BIOS 启动后首先执行的磁盘程序。这个程序负责加载真正的操作系统。由于这个职能,它可以为内核传递参数,管理多个操作系统的启动,查看基本的硬件信息,识别分区、操作磁盘,还可以提供更多其他功能。目前常见的 Bootloader 有 Grub 和 U-Boot,例如现在的 grub2 开源 Bootloader 程序已经是模块化的了,除了提供基本的加载操作系统的功能外,每一个模块都是单独存在的,要使用该模块所实现的命令,grub 需要首先加载这个模块。

U-Boot 常用在 Sparc 或 MIPS 系统,x86 中 Grub 用的最多。要注意的是在安装 grub-install 之前要确认安装的 grub 程序是 grub-bios 还是 grub-efi,因为两个是不同的软件包。

如果有裸片编程的经验就很容易理解 Bootloader。硬件启动时首先要设置一些寄存器,在代码执行的时候,硬件就已经让你的 CPU 可以访问到内存了(或者需要首先设置寄存器),并且把硬件映射到了一定的内存空间。这时你只需要编写简单的 REG=0x123 之类的 C 语言代码,就可以按照 Spec 文档完成硬件初始化。并不是每一个系统都需要 Bootloader,但是 Bootloader 的存在使得内核的启动与硬件解耦和,并且可以支持多个操作系统的选择启动,有的甚至还支持网络加载内核。嵌入式系统一般使用解耦和的特性,桌面系统一般看重多操作系统选择的特性。代码如下。

```
grub> set root=(hd0,1)
grub> linux /boot/vmlinuz-4.4.0-63-generic root=/dev/sda1
grub> initrd /boot/initrd.img-4.4.0-63-generic
grub> boot
```

以上是我们手动在 x86 的 grub shell 中启动 Linux 的流程。我们先设置根盘，再设置对应的内核文件和 initrd 文件，然后启动。可以看到，直观上 Grub 系统的最大的用处就是选择启动哪个内核版本。

在嵌入式系统中，例如路由器，Bootloader 通常存在于 Flash 的最开始部分单独分配的一部分空间，然后内核可以边开发边通过 Bootloader 所提供的网口传输功能把内核二进制传输到 Flash 上，从而被 Bootloader 找到并且正确引导。当要发布整个固件时，直接使用 Bootloader 上传可以运行的文件系统，然后将整个 Flash 用读写器读出来并形成一个 BIN 文件，之后这个 BIN 文件就会被批量用于生产时进行烧录。从这个流程可以看出，Bootloader 在很大程度上是便于开发而存在的。内核完全可以不使用 initrd 文件和 Bootloader 而直接自启动，关键问题在于谁能把最早的内核启动代码加载到内存。

### 4.2.2 内核二进制

Linux 内核的制作一般在某个发行版下完成，也就是说要制作一个 Linux，首先要有一个 Linux（或 UNIX）。通常如果在本机运行，不必调用 `make menuconfig` 命令，而是调用 `make localmodconfig` 命令，这样内核代码的脚本会自动检测当前系统中使用的模块，或者使用当前的内核配置来配置新内核，最后自动生成 `.config` 文件。如果内核版本差异过大，接下来的 `make` 命令在执行的过程中会有很多问题需要回答，然后使用 `make modules_install`，`make install` 命令完成本机内核和内核模块的安装。

内核的核心文件是 `vmlinuz`，这是个压缩后的文件，x86 架构编译后位于 `arch/x86/boot` 路径下。除了内核文件外还需要模块文件，模块文件并不是单独存在的。因为各个模块之间有依赖关系或者是要记录哪些模块启动时需要挂载，哪些不需要挂载，这些相关的文件连同模块本身通常放在 `/lib/modules/4.1.2/`（假定是 4.1.2 版本内核）路径下。但这并不是绝对的，而且内核代码的存放位置也不是绝对的，只要 grub 能够指定即可。这些模块也都是在配置内核的时候选择的，一个功能可以选择被编译进内核二进制，也可以选择编译成模块或者选择不编译。若选择了编译成模块（M 选项），`make` 命令就会生成对应的 `ko` 后缀文件，即选择编译成内核模块的二进制文件。

内核的编译都是先进入各个目录，生成 `built-in.o`，然后在上层根据一定的规则

组合生成 `vmlinux`（例如 `arch/arm/kernel/vmlinux.lds`），然后经过处理和压缩得到最终文件。

内核文件的生成：首先 `ld` 命令链接生成的 ELF 文件（`vmlinux`），然后 `strip`（`objcopy`）命令组装自解压组件为压缩后可自解压的 `vmlinuz`（`zImage`）。嵌入式版本的内核和 x86 架构内核的生成可能会不一样，这取决于使用的 Bootloader。例如 U-Boot 使用 `uImage`，这是在 `zImage` 的基础上添加一个 U-Boot 专用的头部。

在这个过程中还要生成 `System.map` 文件。因为 `vmlinux` 文件已经被 `strip` 了，需要一个单独的文件存放符号表，否则内核无法调试。`kdress` 工具可以用来给 `vmlinux` 重新添加 `System.map` 中的符号表，使用这个生成的内核文件配合 `/proc/kcore` 文件就可以用 `gdb` 调试内核了。

内核编译完成安装模块时会同时生成 `modules.dep` 和很多 `map` 文件。这些文件也可以手动生成（内核编译无非是执行了一系列的命令，例如 `depmod`）。这些 `map` 文件（例如 `modules.alias`）定义的是什么样的硬件应该加载本模块，而 `modules.dep` 文件定义的是各个模块之间的依赖关系，即如果要加载本模块则需要预先加载哪些模块。

生成 `modules.dep` 文件的命令是 `moddep`，而开机启动时一次性加载所有需要的模块的代码是 `modprobe`。这个命令可以根据 `modules.dep` 文件的内容加载尽可能多的模块。有的发行版“认为”这是不合理的，于是它们在 `/etc` 下建立了目录结构，启动时只能使用 `insmod` 逐个加载目录结构中定义的模块。

总体来说，什么模块需要加载，什么模块不需要加载，到目前为止还没有一个很好的解决办法。

## 4.3 Linux 的启动原理

一个 Linux 系统永远由 Bootloader、Kernel、文件系统三个元素组成。有的嵌入式 Linux 系统经过适当调整可以不需要 Bootloader，但如果最新的内核也这么做，那么代价还是很大的，所以以后的系统一般都会有 Bootloader。内核的文件系统至少要有有一个，一般会有两个。一个是 `initrd`，另外一个实际运行的根文件系统，当然还可以挂载无数的文件系统。

需要注意的是，由于 EFI 的出现，其功能越来越强大，导致其可能会慢慢实现



Bootloader 的功能，直到完全实现。因此 Bootloader 不会消失，但是可能会与 EFI 合并。

### 4.3.1 Linux 的最小系统制作和启动

思考 Linux 的启动，最好的方法就是自己制作一个最小 Linux 系统。其实方法很简单，只是在制作 Linux 系统时可能会遇到很多问题，下面介绍一下制作 Linux 系统的方法可以让整个过程一目了然。代码如下。

```
#首先制作一个磁盘
dd if=/dev/zero of=disk ibs=4096 count=4096
#在磁盘上创建一个文件系统
mkfs.ext4 disk
#把创建的磁盘文件挂载到目录
mkdir mnt
mount disk mnt/
#创建文件系统的一些目录，并且放入自己想要放入的文件（例如内核）
mkdir sbin bin etc var root
#卸载文件系统
umount mnt
dd if=disk bs=1k | gzip -v9 > rootfs.gz

mount osImage
fdisk /dev/loop1
mkfs /dev/loop1
#然后创建文件系统目录，放入你要放入的程序
#安装 Bootloader
grub-install /dev/loop1
#内核目录：make，拷贝内核 bzImage (arch/x86/boot/下) 到/boot，生成 initrd
#启动时，指定 Grub 内核和 initrd，传递适当的参数，boot
```

创建了一个最小文件系统，但是还没有安装 Grub，以 gz 的格式保存时可以发现，这其实是一个 initrd 类文件系统。下面简单介绍一下如何安装 Grub，代码如下。

```
truncate -s32M rootfs
fdisk rootfs //创建一个分区。使用 fdisk 的 “b,n” 命令，后续都按回车即可（或者你可以创建多个）
grub-install rootfs
```

上述代码有很多种实现的方法，也可以使用 Loop 设备，这个流程可以快速演示一个可启动的文件系统的制作方法。

这里重要的不是制作过程，而是启动过程。我们看到安装了 grub 之后，该磁盘就能确保启动了，因为 grub 软件的启动是不需要操作系统的，而是单独地引导程序。只要磁盘可用，它就能工作。grub 启动后会弹出 grub 的命令行界面。grub1 和 grub2 两个不同版本的命令已经有了很大的变化，我们这里说的是 grub 2 的命令。使用 Linux 命令指定内核所在的目录（如果你发现没有 root，还需要先 set root 命令，但 grub 一般能自动发现），然后使用 initrd 命令指定 initrd 文件系统，输入 boot 后，系统就会启动了。内核发行版越新，想要去掉 initrd 越难。也就是说在 Linux “工业界” 越来越倾向于认可 initrd 文件系统作为系统启动过程中不可或缺的一部分。

### 4.3.2 initrd 文件系统

直接使用 initrd 文件系统其实是不准确的，因为启动过程中的过渡根文件系统可以是 initrd，也可以是 initramfs。这两个根文件系统的内容是一样的，只是组织方式不一样。

首先建立块（可以使用 dd 命令），然后对块文件进行格式化（例如可以格式化为 ext2 文件系统），将所有需要的文件和根文件的目录都建立好，复制了必须的程序进去，这一点两者是一样的。接下来生成 initramfs 的做法是用 cpio 命令打包，然后进行 zip 压缩。而生成 initrd 的方法则是不使用 cpio 命令，直接进行 zip 压缩。

需要注意的是，不要被各个发行版的命名给弄混了。现在的发行版都是 initramfs，之前的发行版是 initrd。但是由于历史原因，有的发行版现在还是会命名为 initrd。两者的不同是修改的方式不同，initrd 文件的类型，如图 4-2 所示。

```
root@ubuntu:/boot# file initrd.img-4.4.0-51-generic
initrd.img-4.4.0-51-generic: gzip compressed data, last modified: Thu Feb 9 15:46:45 2017, from unix
```

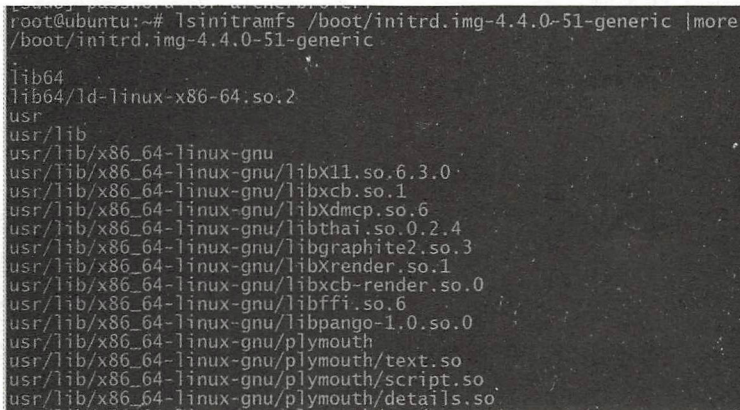
图 4-2

代码如下。

```
cd /boot/
mkdir initrd_fs
cp initrd.img-4.4.0-57-generic initrd_fs/initrd.gz
cd initrd_fs
gunzip initrd.gz
```

修改一个无格式的 img 系统文件使用的方法是 mount，该文件会以 loop 设备的形式存在。而修改一个 cpio 文件的方式是用 cpio 命令归档成目录，不需要 mount。直接修改目录再 cpio 归档后就又是可以用的文件系统了。后者之所以能取代前者，是因为 mount 是一个 root 程序，是系统级的操作，而 cpio 只是一个文件的操作，既方便又安全。

Linux 下的 lsinitramfs (lsinitrd) 命令可以不用 mount 或解压就可以查看 initrd 里的内容。lsinitramfs 文件系统内容，如图 4-3 所示。



```
root@ubuntu:~# lsinitramfs /boot/initrd.img-4.4.0-51-generic | more
/boot/initrd.img-4.4.0-51-generic
lib64
lib64/ld-linux-x86-64.so.2
usr
usr/lib
usr/lib/x86_64-linux-gnu
usr/lib/x86_64-linux-gnu/libx11.so.6.3.0
usr/lib/x86_64-linux-gnu/libxcb.so.1
usr/lib/x86_64-linux-gnu/libxdmcp.so.6
usr/lib/x86_64-linux-gnu/libthai.so.0.2.4
usr/lib/x86_64-linux-gnu/libgraphite2.so.3
usr/lib/x86_64-linux-gnu/libXrender.so.1
usr/lib/x86_64-linux-gnu/libxcb-render.so.0
usr/lib/x86_64-linux-gnu/libffi.so.6
usr/lib/x86_64-linux-gnu/libpango-1.0.so.0
usr/lib/x86_64-linux-gnu/plymouth
usr/lib/x86_64-linux-gnu/plymouth/text.so
usr/lib/x86_64-linux-gnu/plymouth/script.so
usr/lib/x86_64-linux-gnu/plymouth/details.so
```

图 4-3

initrd 是一个过渡文件系统，里面就有程序和数据。而所有的程序和数据都是为了它存在的目的服务的，那么它存在的目的是什么呢？我们可以看一下内核启动了 initrd 里面的什么程序来得出结论。

内核启动 initrd 里的第一个程序是/init，这是内核“写死的”，想要改变就修改内核代码。这里介绍一种典型的情况，这个 init 程序是一个脚本文件，任务流程如下（这只是一个案例，各个版本之间可能会有比较大的变化）。

(1) 建立一个 sysroot 根目录。该目录用于挂载之后要启动的真实文件系统。



(2) 设置命令的环境变量, 这样后面的命令就都可以在环境变量指定的目录中搜索到了。

(3) mount proc 文件系统到/proc, 这是 Linux 内核动态状态的一个表现和设置入口。

(4) mount sysfs 到/sys, 这是 Linux 内核资源的有组织的表现和设置入口。

(5) mount devtmpfs 到/dev, 这是临时表征当前设备节点的文件系统, 可以加速系统的启动 (/dev 目录对当前用户程序的运行至关重要)。

(6) 准备/dev 目录中的一些节点 (例如 stdin、stdout、stderr 等 fd, 这里的 fd 是指文件描述符。对内核中已有的静态节点进行创建)。

(7) 提供为内核输入额外参数的机会。

(8) 解析内核启动参数并执行 (例如在 udev 开始之前执行一些准备), 由此可以看出内核参数并不一定都是给内核来解析的。

(9) 启动 systemd-udevd, 并配置其要响应的行为。

(10) 循环处理\$hookdir/initqueue 下的任务。

(11) mount 根文件系统。

(12) 找到根文件系统的 init 程序。

(13) 停止 systemd-udevd 服务程序。

(14) 做一些清理操作。

(15) 切换到根文件系统的 init 程序, 启动完成。

可以看出整个 initrd 文件系统存在的目的就是挂载根文件系统。所以当根文件系统可以直接挂载的时候就不需要了。但是现在越来越复杂的网络和设备让这个 initrd 的存在变得非常有必要。可以看出, 嵌入式系统是万万不需要 initrd 文件系统的。例如 SCSI 设备内核就很难识别, 但是现在很多存储设备已经是 SCSI 的了。因此, 也可以将 SCSI 编译进内核, 让内核可以直接识别。

### 4.3.3 EFI 启动桩

由于 EFI 的强大功能, 现在的 Linux 越来越倾向于不使用 Bootloader, EFI 启动桩就是一个大胆的尝试, 其在压缩后的 Linux 内核的前面加上了一段可执行程序, 完成 Bootloader 的工作。但是目前该功能还比较弱, 对 Linux 的磁盘的识别和设备驱动的加载也有很大问题。因此, 这代表一个发展方向, 但还不至于快速淘汰 Bootloader。

### 4.3.4 启动管理程序

我们知道系统第一个启动的进程是 `init` 进程，但 `init` 进程不仅位于用户空间，而且位于内核代码的进程。确切地说，在系统启动过程中内核中的 `init` 进程被用户空间的 `init` 进程替换执行。在内核中 `init` 执行完操作后，一般会调用初始化系统来初始化整个系统的应用程序或者服务器。这个初始化系统最原始的是 `linuxrc` 脚本，当然，包括这个脚本之后的所有脚本都是 Linux 操作系统的用法，并不是内核的规定。也就是说你可以自由地指定任何脚本的执行顺序和定义不同脚本的意义，而不用修改内核代码。

常用的 Linux 启动脚本一般有 `/linuxrc`、`/etc/rcS`、`/etc/rc.local`、`/etc/profile`。这些脚本都是可有可无的，如果有需要就一个一个地在脚本中先后调用。必须要注意的是，这些脚本连同名字和路径都是可以随意定制的，不同的发行版可能会选择不同的位置和顺序，但是这几个名字的通用性是 Linux 操作系统长期演化的结果。

只是脚本无法满足健壮的系统对于启动和服务的管理需求，因为不能要求所有对服务的管理都通过写脚本完成。为此 Linux 的一般做法是生成一个专门的目录，想要启动的服务就生成一个规定格式的文件放到目录中，Linux 发行版一般都有写脚本去遍历执行整个目录的服务，早期常见的这种系统是 `rcN.d` 目录，`N` 一般取值 0~6，例如执行 `init 6` 命令，就会执行 `rc6.d` 目录中的脚本以关闭和打开对应的服务。对于这些被定义的服务，甚至通过复杂的语法限制还可以实现精细的启动顺序控制 (`systemd`)。

在嵌入式系统中，一般不使用 `systemd` 启动管理系统，而是直接采用 `rcS` 等启动脚本。一个内核 `init` 程序的示例代码如下。

```
//init/main.c
asmlinkage __visible void __init start_kernel(void)
{
    //本函数此处往上省略
    rest_init();
}
static noinline void __ref rest_init(void)
{
    int pid;
    rcu_scheduler_starting();
```

```
kernel_thread(kernel_init, NULL, CLONE_FS);
//本函数代码此处往下省略
}
static int __ref kernel_init(void *unused)
{
    int ret;
    kernel_init_freeable();
    async_synchronize_full();
    free_initmem();
    mark_readonly();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();
    rcu_end_inkernel_boot();
    if (ramdisk_execute_command) {
        ret = run_init_process(ramdisk_execute_command);
        if (!ret)
            return 0;
        pr_err("Failed to execute %s (error %d)\n",
               ramdisk_execute_command, ret);
    }
    if (execute_command) {
        ret = run_init_process(execute_command);
        if (!ret)
            return 0;
        panic("Requested init %s failed (error %d).",
              execute_command, ret);
    }
    if (!try_to_run_init_process("/sbin/init") ||
        !try_to_run_init_process("/etc/init") ||
        !try_to_run_init_process("/bin/init") ||
        !try_to_run_init_process("/bin/sh"))
        return 0;
    panic("No working init found. Try passing init= option to kernel."
          "See Linux Documentation/admin-guide/init.rst for guidance.");
}
```

可以按照上述流程追踪内核的启动代码。我们从底层 boot 之后的内核正式启动



入口。从 `start_kernel` 开始，我们能够发现其最终调用到了 `kernel_init`，在这个函数里调用了内核“写死的”约定启动程序 `/sbin/init`、`/etc/init`、`/bin/init`、`/binsh`。在这之前还会尝试首先调用 `initrd` 里面的 `init`。

### 1. Sys V init:runlevel

系统中有很多服务，人们在 Linux 出现的时候就在想如何管理这些服务了。问题是有些服务需要启动，有些不需要启动。登录图形界面需要某一些服务，但不登录图形界面就不需要这些服务，单用户登录可能需要的服务更少，如此就需要差别化地启动所需要的服务。这些服务的组织早期的方法是使用 `rcN.d` 目录，这一整套目录的约定就是 Sys V init 的 `runlevel` 规范。

### 2. 针对网络服务

由于大部分 Linux 操作系统上运行的后台服务进程都是在监听某一个端口，Windows 操作系统的做法是要使用什么就打开什么进程，在内存里一直监听睡眠等待。Linux 操作系统认为既然都是监听网络服务，找一个超级进程监听全部的端口，哪个端口有数据来再启动哪个程序，这样节省内存的思想就使得 `xinetd` 守护进程诞生了。

`xinetd` 管理监听所有的端口，当该端口有请求到达的时候启动对应的端口处理服务进程，导致的结果是初次响应变慢。而且启动了对应的服务之后没有请求，是不是要关闭该进程呢？否则只要启动了就永远启动。关闭进程后又打开，岂不是更耗费系统资源？由于大部分系统管理员清楚自己的系统要提供什么服务，所以 `xinetd` 用得越来越少。

### 3. 开机启动所有需要启动的进程

Linux 从 UNIX 演化而来，所以最初的启动管理程序是在 UNIX 的 `init` 基础上创新的。定义的启动服务程序是 `runlevel` 机制，系统默认会定义 6 种或更多的 `runlevel`，每一种 `runlevel` 都会启动不同的程序。例如约定的 1 是只启动单用户无图形界面；3 是多用户无图形界面；5 是多用户图形界面；0 是关机；6 是重新启动。通常默认的启动 `runlevel` 值放在 `/etc/inittab` 文件中。`init` 进程通过读取这个文件获得 `runlevel` 值，然后去对应的 `/etc/rc3.d` 等不同的目录下执行在该目录定义的属于这个运行级别的程序。启动之后可以通过 `/sbin/telinit` 程序改变运行级别。

#### 4. upstart

为了克服 `init` 的同步顺序启动带来的效率低下的困难，`upstart` 实现了异步事件驱动的启动模式，在某些情况下提高了系统的启动速度。由于 `upstart` 对 `init` 进程的提升和采用的异步机制，使得开机启动速度和组织有了更多的改善空间，有很多发行版采用了。但是由于 `systemd` 的迅速崛起，很多采用 `upstart` 的系统也迅速切换到 `systemd`，`upstart` 作为一个过渡版本的启动管理程序已基本退出历史舞台。

#### 5. systemd

`systemd` 起源于 Tizen（由 Intel 和三星研发的 Linux 操作系统），经过完善和丰富形成了现在的程序集。起初 `systemd` 设计只是用来取代 `init` 和 `startup` 的，但在逐渐开发的过程中其功能越来越丰富，没有一个发行版不愿意接受如此强大的质量优秀的开源代码。传统的 `init` 开机启动进程是顺序的，并且由 Shell 脚本执行很多开机指令来完成系统的初始化。`systemd` 将启动尽可能地并行化，并且将很多本应由 Shell 执行的逻辑移到 `systemd` 程序中来，提高执行速度。

目前 `systemd` 除了对启动进行管理，还对用户端封装了几乎所有的系统服务，只是很多系统服务还没有被广泛应用。例如原来的 `cron` 被 `systemd` 的调度执行部分取代，`udev` 被其 `device hotplugging` 取代。为了向上提供 `ipc`，`systemd` 还封装提供了 UNIX Domain Socket 和 D-Bus 给其他服务程序。其用统一的原语实现了几乎整个 Linux 系统服务，并且提供对其他外置服务的支持（其本身就是作为服务管理程序存在的）。

`systemd` 进程是系统的第一个启动进程，也是系统的最后一个结束进程，是所有用户端进程的根进程。其比传统的 `init` 在处理子进程上有很多改进，例如可以支持进程关闭后自动重启（用 `init` 的 `respawn` 语义也可以），不产生僵尸进程等。

为了启动的并行化，`systemd` 定义了一整套脚本语义。所有要启动的进程服务都要使用其规定的语义完成 `unit` 文件。在 `init` 系统中，每个进程都是由各自的独立脚本完成启动的，在 `systemd` 的 `unit` 文件中，`service`、`socket`、`device`、`mount`、`automount`、`swap`、`target`、`path`、`timer`（替代 `cron`）、`snapshot`、`slice` 和 `scope` 这 12 种语义可以定义丰富的启动信息。还有一个 `target` 的概念，一个 `target` 就是一群 `Unit` 的集合，也就是一个启动组。

`systemd` 是守护进程；`systemctl` 用来定义 `systemd` 的服务和行为；`systemd-analyze` 用来分析启动的效率。`systemd` 目前完全使用内核的 `cgroup` 接口进行开发，所以 `cgroup` 也变成了现代 Linux 操作系统的标配。代码如下。



```
# 重启系统
$ systemctl reboot
# 关闭系统, 切断电源
$ systemctl poweroff
# CPU 停止工作
$ systemctl halt
# 暂停系统
$ systemctl suspend
# 让系统进入冬眠状态
$ systemctl hibernate
# 让系统进入交互式休眠状态
$ systemctl hybrid-sleep
# 启动进入救援状态 (单用户状态)
$ systemctl rescue

# 列出正在运行的 Unit
$ systemctl list-units
# 列出所有 Unit, 包括没有找到配置文件的或者启动失败的
$ systemctl list-units --all
# 列出所有没有运行的 Unit
$ systemctl list-units --all --state=inactive
# 列出所有加载失败的 Unit
$ systemctl list-units --failed
# 列出所有正在运行的, 类型为 service 的 Unit
$ systemctl list-units --type=service
# 查看 nginx.service 这个 unit 的启动依赖关系
$ systemctl list-dependencies --all nginx.service

# 查看启动耗时
$ systemd-analyze
# 查看每个服务的启动耗时
$ systemd-analyze blame
# 显示瀑布状的启动过程流
$ systemd-analyze critical-chain
# 显示指定服务的启动流
$ systemd-analyze critical-chain atd.service
```



比较重要的 systemd 服务有以下 6 个。

- **console**: 取代传统的虚拟终端。
- **journald**: 取代传统的 syslog、syslog-ng、rsyslog。
- **logind**: 取代传统的用户登录服务 (ConsoleKit、gnome-session)。
- **networkd**: 取代传统的网络配置 (如 Network Manager)。
- **timedated**: 所有与时间有关的操作都将在此集成。
- **udev**: udev 的代码被 systemd 完全吸收合并。

基于 systemd 的成功, 很多人希望继续创新, 试图更好地使用或取代 systemd。比较好的成果有 endev、uselessd、systemdbsd、console kit2, 但是都没有真正地对 systemd 构成威胁。

### 4.3.5 Linux 内核启动顺序

Bootloader 将内核加载到内存后, 需要将控制权交给内核。第一步是要解压内核, 由于内核是自解压的, 解压的入口相关文件是 arch/arm/boot/compressed/head.S。完成解压后就是搬运, 因为解压并不把内核放在其最终执行的位置, 移动之前还要进行保存可能被其覆盖的代码的操作。

第二阶段从 arch/arm/kernel/head.S 开始, 是内核的实际功能地点。主要完成的工作有 CPU ID 检查、machine ID 检查、创建初始化页表、设置 C 语言代码运行环境、跳转到内核第一个真正的 C 函数 startkernel 开始执行。

第三阶段从 start\_kernel 开始完全是 C 语言了。其首先用大内核锁锁住内核, 保证独占, 然后调用平台相关的初始化操作 (arch/arm/kernel/setup.c 里的 setup\_arch()), 在这里内核启动后可使用的所有内存被初始化。被初始化的还有页表结构、MMU、中断、内存区域、计时器、Slab、VFS 等。所以, 如果要预留不被内核使用的内存空间, 应该在这里预留, 然后跳到 init 内核进程 (不是用户空间的 init 进程)。

第四阶段就是内核的 init 进程将自身替换为用户端的 init 进程进行执行。

# 5

## 第 5 章

# 进程

### 5.1 进程原理

#### 5.1.1 服务与进程

进程是满足用户需求的一系列正在执行的任务，有的为了提供一个交互的界面；有的为了提供一个后台的演算；有的为了提供一个网络服务；有的为了利用磁盘资源做存储等。归根结底，进程就是需求的承载体。PC 本身就是通用化的设备，自然所有的 PC 系统都要满足各种各样的需求。因此，提供一个直观的进程模型就是各种服务实现的基础。

进程在种类上对用户来说千奇百怪，有的服务于医疗；有的服务于电影娱乐；有的服务于游戏。这些进程在用户看来是按行业区分的，但是在操作系统看来，所有这些进程都是对操作系统所管理的硬件资源的请求。有的需要显示器；有的需要磁盘；有的需要 CPU；有的需要网络。所以，在用户看来进程是一个个不同应用种类的服务，但是在操作系统看来，是侧重于不同种类资源请求的进程。

### 5.1.2 资源与进程

进程对资源的需求大体分为磁盘 IO 密集型、网络 IO 密集型、内存密集型、CPU 密集型和显卡密集型。这 5 种类型也是 PC 所能提供的 5 个主要的资源。PC 资源和世界资源一样，永远是有限的，关键是找到合理分配资源的方式。世界由不同的国家构成，世界资源的分配是一个博弈的过程。而 PC 可以只有一个主控系统，所以可以更加合理地充分地利用资源。但是目前实现的调度算法远远称不上优秀，在使用系统时几乎不能被感知到，除非你使用的进程耗尽了几乎 100% 的 CPU 时，才会把精力放在对调度系统的优化上。

PC 有个特点，就是无论是哪种服务都需要有 CPU 的参与，而对其他资源的依赖很多都是非独占的，例如几个进程一起使用内存、网络和显卡等。除非你有多个 CPU，否则你永远无法让一个 CPU 同时为多个进程服务。即使你有多个 CPU，其数量也不会比进程多。所以 CPU 的时分复用就很重要，其他资源由于其并行性，其资源管理器提供的都是申请服务模型的服务，而只有 CPU 的资源管理器主动发起调度。所以，必须要理解我们常说的进程调度都是指调度 CPU，因为只有 CPU 才最需要调度。但是随着各个服务的复杂化，对于 IO 或者显卡的请求也越来越呈现无法充分并行的情况，也就是资源不足，而 CPU 也越来越具有并行能力，即 CPU 资源逐渐充足。所以未来的调度进程也不一定只会以 CPU 的时间片作为调度单元。真理是：永远以瓶颈资源作为调度的核心。可能有那么一天，CPU 不是问题（现在有 24 核的服务器，一般不会在使用 CPU 时遇到瓶颈），最大的瓶颈在网络，在于各个进程都同时对有限的网络带宽提出需求。这时 CPU 可能就会被作为服务的提供方，网络带宽的调度程序就变成了进程调度的核心话题了。请求满足模型与调度系统是同一个功能的两种不同视角的实现。

### 5.1.3 进程概念

进程这个概念最早是不存在的，用现在的话来说进程就是单任务的操作系统。但在以前人们认为一个在板子上跑起来的软件只有一条执行流水线是很正常的事情，但是随着业务逻辑越来越复杂，人们对一个板子同时做多个事情的要求也越来越大，于是人们想办法同时模拟出多个正在执行的代码段。之所以说模拟，是因为以前 CPU 着实只有单核，其同时执行多个代码就超出了物理限制。因此，只能切分



CPU 的时间片造成假并行的现象。即使是现在有了物理的多核，CPU 的时间片也会被切分用以创造超过 CPU 个数的并行现象。C++ 的 Boost 库中的协程并不需要多个并行的调度实体，调度是在库中模拟完成的。

对多任务需求的解决方案带来了有利和不利结果。有利的是满足了这个编程需求，这个需求非常大，大部分新的实际应用如果没有这个需求根本不能实现，甚至没有这个需求的推动就没有现代的计算机技术。但同时带来的副作用是，无论需求多么大，只能想办法克服，毕竟需求是技术生存的原动力。还有两个最大的副作用是资源竞争与执行实体的调度。

要在一个只能运行一个代码流水线的 CPU 模拟运行多个代码流水线，通过设计上层概念，然后分时分配 CPU 资源，所以 CPU 被人们称为资源。设计的上层概念有很多：进程、线程、workqueue、tasklet、softirq，这还仅仅是在 Linux 中目前仍在使用的代码流水线的概念。在当前的 Linux 内核中，进程与线程几乎是调度一致的，workqueue、tasklet、softirq 等和进程一起参与调度算法的调度，因为不调度就意味着不能被 CPU 执行。这些代码流的概念服务于不同的用途，例如 softirq 和 tasklet 一般用于中断；workqueue 一般用于驱动。而它们被调度的方法通常是一个专用的内核线程在后台执行。进程和线程的概念则一般用于用户空间。而在实际的实现上，softirq、tasklet 或 workqueue 都可以被封装到某个线程，如此调度算法在调度的时候只需要认识线程一种结构就好了，精简算法逻辑是有好处的。但是这也仅仅是实现的方式。完全可以让调度算法认识各种不同的代码流概念，从而在调度时区别对待。

我们现在写用户空间的代码时只认识进程和线程，因为用户空间是内核的接口产品，用户空间的程序员对整个世界的认识。例如用户空间看到进程和线程时，大部分程序员都能说出进程与线程的区别。但是在内核空间，线程与进程没有太大的区别（需要理解进程组的概念），在用户空间看到的它们之间的区别其实是被伪造出来的。

既然进程的概念最后“胜出”了，也就是说为了说明现代的进程，必须要说明实现进程所必须付出的代价：进程调度、资源竞争、进程通信、进程关系以及进程概念是如何被制造出来的。

### 5.1.4 父子关系

进程在 Linux 中被组织为父子关系，这为管理带来了一定程度的方便，也为编

程带来了一些复杂性。这是一个既被人喜欢又被人讨厌的特性。

子进程退出，父进程要调用 `wait` 或 `waitpid` 函数等待回收子进程的资源，否则子进程就一直以“僵尸”状态存在。这就在业务上带来了不便，例如父进程希望启动子进程后继续执行自己的任务，但是又不得不阻塞调用 `wait` 或者 `waitpid` 等待子进程的退出，此时就会带来困难，虽然 `wait` 会响应信号，用 `sigalarm` 能给 `wait` 设置超时，但是这样又会带来其他问题。一个偷懒的做法是用 `signal(SIGCHLD, SIG_IGN)` 来忽略子进程的信号，从而把这个回收工作交给 `init` 进程，但是这么做子进程就脱离了掌控，将无法有效地掌握子进程的状态。也可以考虑使用非阻塞的 `wait`，那么就需要设计一个轮询的机制。

我们在实际的工程中经常遇见的需求是：启动一个子进程，阻塞运行后退出。这时我们最常用的方法是使用 `system` 系统调用（这是在严肃工程中最不建议使用的方法）。这个系统调用通常用来执行一个外部的命令，其内部本质上是首先使用 `fork` 复制一个子进程，然后子进程 `execl` 调用具体的命令来覆盖当前的进程内存，然后 `waitpid` 阻塞等待。这个接口虽然方便，但是会有诸多的问题。比如子进程完整地继承了父进程的信号和 `socket` 等信息；如果在父进程已经使用 `signal(SIGCHLD, SIG_IGN)`，那么在子进程结束时，子进程的返回值不能被 `waitpid` 接收。最重要的问题就是 `system` 使用的是重量级的 `fork` 系统调用，完整地拷贝当前的父进程。类似的封装式的解决方案还有 `popen`，通过打开管道来调用外部命令。`popen` 内部也使用 `fork` 系统，而 `fork` 使用 `clone` 系统调用，所以如果父进程太大就容易出现内存不足的情况（实际情况与 `man` 手册中说的情况有所出入）。手册中说的是当内存不使用 `fork` 系统时就不会出现内存不足的情况，然而在实际的使用中，有时候会出现内存不足的情况。所以，`popen` 和 `system` 一般会使用 `vfork` 实现。

这部分的封装代码是 `stackoverflow` 上的某位网友的作品，笔者一直在使用他的修改版，非常稳定。在实际的使用中，很多时候只需要一次打开一个子进程，其中的一些逻辑就可以删除，但是这样也可以直接使用。要实现这样的封装，首先要获得最大的可打开的 `fd` 数目，示例如下。

```
static long openmax = 0;
#define OPEN_MAX_GUESS 1024
long open_max(void)
{
    if (openmax == 0) { /* first time through */
```



```

    errno = 0;
    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0) {
        if (errno == 0)
            openmax = OPEN_MAX_GUESS;    /* it's indeterminate */
        else
            printf("sysconf error for _SC_OPEN_MAX");
    }
}
return(openmax);
}

```

上述函数可以获得当前系统支持的最大 fd 打开数目。在很多情况下自己写的系统不需要如此检测，或者每次只会打开一个子进程，然后等待其退出，就可以简化这部分逻辑。进程启动逻辑如下。

```

static pid_t    *childpid = NULL; /* 指向运行时生成的子进程数组*/
static int      maxfd; /*open_max 得到的最大 fd 支持数*/
FILE *vpopen(const char* cmdstring, const char *type)
{
    int pfd[2];
    FILE *fp;
    pid_t pid;
    if((type[0]!='r' && type[0]!='w')||type[1]!=0)
    {
        errno = EINVAL;
        return(NULL);
    }
}

```

/\* 第一次调用 vpopen 的时候需要初始化全局静态数组，这一步可以根据自己情况进行删除和修改 \*/

```

    if (childpid == NULL) {
        maxfd = open_max();
        if ( (childpid = (pid_t *)calloc(maxfd, sizeof(pid_t))) == NULL)
            return(NULL);
    }
    //pipe 系统调用接受一个长度为 2 的 fd 数组，一个用于输入，一个用于输出
    if(pipe(pfd)!=0)    {
        return NULL;
    }
}

```



```

    if((pid = vfork())<0)
    {
        return(NULL);
    }
else if (pid == 0) {    /* 进入子进程*/
//由于管道的特点是只能读或者只能写，所以选择了读就得关闭写
    if (*type == 'r')
    {
        close(pfd[0]);
        if (pfd[1] != STDOUT_FILENO) {
            dup2(pfd[1], STDOUT_FILENO);
            close(pfd[1]);
        }
    }
else
    {
        close(pfd[1]);
        if (pfd[0] != STDIN_FILENO) {
            dup2(pfd[0], STDIN_FILENO);
            close(pfd[0]);
        }
    }
/* 当打开一个新的子进程时，关闭之前打开的所有子进程 */
for (int i = 0; i < maxfd; i++)
    if (childpid[ i ] > 0)
        close(i);
//执行实际的命令
    execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
    _exit(127);
}
//根据读还是写的管道方向，将系统的 fd 转变为 C 的 FILE 指针
    if (*type == 'r') {
        close(pfd[1]);
        if ( (fp = fdopen(pfd[0], type)) == NULL)
            return(NULL);
    } else {
        close(pfd[0]);

```

```

        if ( (fp = fdopen(pfd[1], type)) == NULL)
            return(NULL);
    }
    //记录子进程的 pid, 以方便之后打开新的子进程的时候关闭已经打开的子进程
    childpid[fileno(fp)] = pid;
    return(fp);
}

```

我们对比打开操作与系统的 pipe 调用的区别, 就能发现有两个显著的不同: 一是使用了 `vfork`, 虚拷贝不会造成内存不足的问题和其他的继承问题; 二是每次调用 `vpopen` 时都要关闭之前调用打开的子进程, 这个功能可以根据需要删除。关闭逻辑如下。

```

int vpclose(FILE *fp)
{
    int    fd, stat;
    pid_t  pid;
    if (childpid == NULL)
        return(-1);    /* popen() 没有打开过就不执行/
    fd = fileno(fp);
    if ( (pid = childpid[fd]) == 0)
        return(-1);    /* 管道的 fd 没有被打开*/
    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);
    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1);
    return(stat);
}

```

可以看到关闭子进程的时候就是关闭打开的文件句柄和 `waitpid` 阻塞等待, 所以如果要运行一个命令而不需要结果, 完全可以直接调用 `vpopen`, 然后直接调用 `vpclose`, 程序就会一直阻塞到子进程退出, 有 `pthread` 的 `join` 的效果。这个封装用于替代 `popen` 和 `system`, 用户也不必自己使用 `vfork`、`exec`, 以及 `clone` 系列底层命令编码进程的启动。优化 `waitpid` 的过程如下。



```

uint64_t wait_time = 0;
while(true){
    int result = waitpid(pid, &stat, WNOHANG);
    if(result < 0){
        return -1;
    }else if(result == 0){
        sleep(1);
        wait_time++;
        if(wait_time > conf.plugin_max_run_time){
            if(pid > 0){
                killpg(pid, 9);
            }else{
                cerr<<"pg pid is less than 0:"<<pid;
            }
        }
    }else{
        break;
    }
}

```

一般情况下，使用 `waitpid` 操作不会那么顺利，所以应根据实际情况做出如上的修改。这样当 `waitpid` 因为其他原因退出时，不至于错过子进程的回收时机，能够避免子进程 `defunct`。还添加了允许子进程运行最长时间的逻辑。对应的可以在 `vpopen` 中生成的子进程中添加，进程组代码如下。

```

if(setpgid(0,0) < 0){
    cerr<<"child " << pid<<" setpgid failed:"<<strerror(errno);
}

```

如此将生成的子进程以及孙子进程都纳入同一个进程组，并且与父进程不一样，如果父进程在等待子进程时超时，就可以直接强制关闭整个子进程组，而不影响父进程本身。

### 5.1.5 ptrace 系统调用

`ptrace` 系统调用可以强制让一个进程成为另外一个进程的父进程，并且可以深



入的对子进程进行流程控制。Linux 下有几个常用的工具是基于 ptrace 系统调用的，大部分用来做普通的程序分析，尤其是内存数据审查和修改。ptrace 这个系统调用的功能大致分为两部分：一部分是用来控制调试进程的；另一部分是查看和修改被调试进程的数据的。该系统功能所以几乎涵盖了所有调试正在运行进程的需求。

由于 ptrace 是以线程为单位的，也就是说如果一个进程有多个线程，父进程就需要分别 ptrace 所有的线程，如果多线程的程序在设计的时候没有考虑被 ptrace，那么在使用 ptrace 时就要非常小心，一个暂停可能就会影响进程整体的执行。例如一个没有被 ptrace 的线程在不断地向缓存里写数据，由于你的 ptrace 调用可能导致 ptrace 的消费者线程运行速度变慢，导致缓存爆炸。下面给出一个完整的 ptrace 示例程序，读者可以在机器上直接编译，可逐步修改以理解其中的含义。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <signal.h>
#include <elf.h>
#include <sys/types.h>
#include <sys/user.h>
#include <sys/stat.h>
#include <sys/ptrace.h>
#include <sys/mman.h>
#include <sys/wait.h>
#include <iostream>
using namespace std;

typedef struct handle {
    Elf64_Ehdr *ehdr;
    Elf64_Phdr *phdr;
    Elf64_Shdr *shdr;
    uint8_t *mem;
    char *symname;
    Elf64_Addr symaddr;
    struct user_regs_struct pt_reg;
```

```

    char *exec;
} handle_t;
int global_pid;
Elf64_Addr lookup_symbol(handle_t *, const char *);
char * get_exe_name(int);
void sighandler(int);
#define EXE_MODE 0
#define PID_MODE 1

int main(int argc, char **argv, char **envp) {
    int fd, mode, c;
    handle_t h;
    struct stat st;
    long orig;
    int status, pid;
    char * args[2];
    printf("Usage: %s [-ep <exe>/<pid>]    [-f <func name>]\n", argv[0]);
    while ((c = getopt(argc, argv, "p:e:f:")) != -1) {
        switch(c) {
            case 'p':
                pid = atoi(optarg);
                h.exec = get_exe_name(pid);
                if (h.exec == NULL) {
                    printf("Unable to retrieve executable path for pid: %d\n",
pid);

                    exit(-1);
                }
                mode = PID_MODE;
                break;
            case 'e':
                if ((h.exec = strdup(optarg)) == NULL) {
                    perror("strdup");
                    exit(-1);
                }
                mode = EXE_MODE;
                break;
            case 'f':

```



```
        if ((h.symname = strdup(optarg)) == NULL) {
            perror("strdup");
            exit(-1);
        }
        break;
    default:
        printf("Unknown option\n");
        break;
    }
}

if (h.symname == NULL) {
    printf("Specifying a function name with -f option is
required\n");
    exit(-1);
}

if (mode == EXE_MODE) {
    args[0] = h.exec;
    args[1] = NULL;
}

if ((fd = open(h.exec, O_RDONLY)) < 0) {
    perror("open");
    exit(-1);
}

if (fstat(fd, &st) < 0) {
    perror("fstat");
    exit(-1);
}

h.mem = (uint8_t*)mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd,
0);

if (h.mem == MAP_FAILED) {
    perror("mmap");
    exit(-1);
}

h.ehdr = (Elf64_Ehdr *)h.mem;
h.phdr = (Elf64_Phdr *) (h.mem + h.ehdr->e_phoff);
```



```

        h.shdr = (Elf64_Shdr *) (h.mem + h.ehdr->e_shoff);
        if (h.mem[0] != 0x7f || strncmp((const char *)&(h.mem[1]), "ELF", 3))
        {
            printf("%s is not an ELF file, mem0: %x, head
str: %s\n", h.exec, h.mem[0], (char *)&(h.mem[1]));
            exit(-1);
        }
        if (h.ehdr->e_type != ET_EXEC) {
            printf("%s is not an ELF executable\n", h.exec);
            exit(-1);
        }
        if (h.ehdr->e_shstrndx == 0 || h.ehdr->e_shoff == 0 ||
h.ehdr->e_shnum == 0) {
            printf("Section header table not found\n");
            exit(-1);
        }
        close(fd);

        if (mode == EXE_MODE) {
            if ((pid = fork()) < 0) {
                perror("fork");
                exit(-1);
            }
            if (pid == 0) {
                if (ptrace(PTRACE_TRACEME, pid, NULL, NULL) < 0) {
                    perror("PTRACE_TRACEME");
                    exit(-1);
                }
                execve(h.exec, args, envp);
                exit(0);
            }
        }
        else { // attach to 'pid'
            if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
                perror("PTRACE_ATTACH");
                exit(-1);
            }
        }
    }
}

```

```

wait(&status);
    global_pid = pid;
    printf("Beginning analysis of pid: %d at %lx\n", pid, h.symaddr);
    if ((orig = ptrace(PTRACE_PEEKTEXT, pid, h.symaddr, NULL)) < 0) {
        perror("PTRACE_PEEKTEXT");
        exit(-1);
    }
    long trap = (orig & ~0xff) | 0xcc;
    if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, trap) < 0) {
        perror("PTRACE_POKETEXT");
        exit(-1);
    }
    trace:
    if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0) {
        perror("PTRACE_CONT");
        exit(-1);
    }
    wait(&status);
    if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGTRAP) {
        if (ptrace(PTRACE_GETREGS, pid, NULL, &h.pt_reg) < 0) {
            perror("PTRACE_GETREGS");
            exit(-1);
        }
        printf("\nExecutable %s (pid: %d) has hit breakpoint 0x%lx\n",
h.exec, pid, h.symaddr);
        printf("%rcx: %llx\n%rdx: %llx\n%rbx: %llx\n"    "%rax:
%llx\n%rdi: %llx\n%rsi: %llx\n"    "%r8: %llx\n
%r9: %llx\n%r10: %llx\n"    "%r11: %llx\n%r12 %llx\n%r13 %llx\n"
"%r14: %llx\n%r15: %llx\n%rsp: %llx",    h
        .pt_reg.rcx, h.pt_reg.rdx, h.pt_reg.rbx,    h.pt_reg.rax, h.pt_reg.rdi,
h.pt_reg.rsi,    h.pt_reg.r8, h.pt_reg.r9, h.p
        t_reg.r10,    h.pt_reg.r11, h.pt_reg.r12, h.pt_reg.r13,    h.pt_reg.r14,
h.pt_reg.r15, h.pt_reg.rsp);
        printf("\nPlease hit any key to continue: ");
        getchar();
        if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, orig) < 0) {
            perror("PTRACE_POKETEXT");

```



```

        exit(-1);
    }
    h.pt_reg.rip = h.pt_reg.rip - 1;
    if (ptrace(PTRACE_SETREGS, pid, NULL, &h.pt_reg) < 0) {
        perror("PTRACE_SETREGS");
        exit(-1);
    }
    if (ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL) < 0) {
        perror("PTRACE_SINGLESTEP");
        exit(-1);
    }
    wait(NULL);
    if (ptrace(PTRACE_POKETEXT, pid, h.symaddr, trap) < 0) {
        perror("PTRACE_POKETEXT");
        exit(-1);
    }
    goto trace;
}
if (WIFEXITED(status))    printf("Completed tracing pid: %d\n", pid);
exit(0);
}

Elf64_Addr lookup_symbol(handle_t *h, const char *symname) {
    int i, j;
    char *strtab;
    Elf64_Sym *symtab;
    for (i = 0; i < h->ehdr->e_shnum; i++) {
        if (h->shdr[i].sh_type == SHT_SYMTAB) {
            strtab = (char *)&h->mem[h->shdr[h->shdr[i].sh_link].
sh_offset];

            symtab = (Elf64_Sym *)&h->mem[h->shdr[i].sh_offset];
            for (j = 0; j < h->shdr[i].sh_size/sizeof(Elf64_Sym); j++) {
                printf("checking %s with %s\n", &strtab[symtab->st_name],
symname);

                if (strcmp(&strtab[symtab->st_name], symname) == 0) {
                    cout<<"found"<<endl;
                    return (symtab->st_value);
                }
            }
        }
    }
}

```



```

        symtab++;
    }
}
return 0;
}

char * get_exe_name(int pid) {
    char cmdline[255], path[512], *p;
    int fd;
    snprintf(cmdline, 255, "/proc/%d/cmdline", pid);
    if ((fd = open(cmdline, O_RDONLY)) < 0) {
        perror("open");
        exit(-1);
    }
    if (read(fd, path, 512) < 0) {
        perror("read");
        exit(-1);
    }
    if ((p = strdup(path)) == NULL) {
        perror("strdup");
        exit(-1);
    }
    return p;
}

void sighandler(int sig) {
    printf("Caught SIGINT: Detaching from %d\n", global_pid);
    if (ptrace(PTRACE_DETACH, global_pid, NULL, NULL) < 0 && errno) {
        perror("PTRACE_DETACH");
        exit(-1);
    }
    exit(0);
}

```

这里的逻辑无论是对于 `ptrace` 的理解还是对二进制的理解都是非常重要的，二进制的相关知识会在后面的章节详细叙述。

另外 `ptrace` 可以被用于入侵系统做注入，`ptrace` 之前的默认行为可以覆盖 `mmap` 或者 `mprotect` 的权限设置，即使是 `.text` 的 `READONLY` 域也能被写入，但是后来内

核打了 pax 或者 grsec 补丁，就能在一定程度上限制这种行为。使用 ptrace 直接向运行中的进程注册代码是可行的，所以 ptrace 系统调用可以用来攻防。如果要使用 ptrace 的功能，建议首先详细且尽可能地使用 libptrace 库。示例如下。

```
ptrace_context ptc;
if(ptrace_open(&ptc, pid) == -1){
    cout<<"ptrace_open(): "<<ptrace_errmsg(&ptc)<<endl;
    return false;
}
if ( ptrace_close(&ptc) == -1 ) {
    fprintf(stderr, "ptrace_close(): %s\n",
ptrace_errmsg(&ptc));
    exit(EXIT_FAILURE);
}
```

一般使用 libptrace 库作为中间件，可以在一定程度上解决可移植性问题，并且提供了对 mmap 文件的访问封装和一些加载库之类的常用操作的封装。代码也比较简单，使用者完全可以实现自己的 ptrace 库。

还有如下一些其他常用的工具依赖 ptrace 审查进程。

- strace: strace 使用 ptrace 在进程和内核之间做记录，打印进程调用的所有系统调用；
- ltrace: ltrace 更重视库函数的调用分析。strace 和 ltrace 配合就可以是一个不错的动态调用分析系统；
- ftrace: 大部分发行版都没有带这个工具，它可以跟踪一个二进制内部的函数调用，正好与 strace 和 ltrace 互补；
- gdb attach: 可以直接使用 gdb 进行进程 attach。

## 5.2 进程调度

Linux 是个多进程的环境，不但用户空间可以有多个进程，而且内核内部也可以有内核进程。在 Linux 内核中线程与进程在调度时没有太大区别，只是中间多了一层调度域，因此在讨论调度算法的时候称作线程和进程都是一样的。调度器调度的是 CPU 资源，按照特定的规则分配给特定的进程。然后进程去申请或使用硬件或资

源。因此这里涉及以下两方面的问题。

(1) 对于调度器而言：

- 调度程序在运行时，如何确定哪一个程序将被调度使用 CPU 资源？
- 如何不让任何一个进程饥饿？
- 如何更快地定位和响应交互式进程？
- 单个 CPU 只有一个流水线，能否一次调度多个进程，同时使用多个 CPU 的物理资源呢？
- 调度来的 CPU 如何让其释放资源？是任其自己释放还是有相关回收机制？

(2) 对于希望被调度的进程而言：

- 如何定义自己被调度的概率？
- 如何在等待被调度的同时接收信号？
- 如何避免自己希望占用的资源在没有被调度时不被别的进程占用，或者在 SMP 环境下没有与其同时使用同一资源的进程？

### 5.2.1 调度策略

Linux 的调度策略分为分时系统和实时系统两种。Linux 本身不是实时系统，但是本着兼容并包的原则，Linux 也实现了实时系统的接口。在操作系统的发展历史上，也很可能就是这种兼容并包、什么都覆盖的思路，让 Linux 顽强地生存了下来。

对于整个内核而言，调度策略包括 `SCHED_NORMAL`、`SCHED_FIFO`、`SCHED_RR`、`SCHED_BATCH` 这 4 种。而标准的调度策略还有两种 Linux 没有实现，即 `SCHED_IDLE` 和 `SCHED_DEADLINE`。`SCHED_NORMAL` 就是我们最常说的 Linux 默认使用的分时调度策略。

无论是实时的调度策略还是普通的调度策略，优先级都是由数值表示的。普通的静态优先级全部为 0，区别普通调度程序可以使用动态优先级，也就是 `nice` 值。实时调度的程序优先级的 `nice` 值为 1~99，也就是说任何一个实时程序的优先级都高于普通程序。所以，内核由 `nice` 值、优先级和具体的调度算法综合作用来决定调度什么进程执行。

当使用 `SCHED_RR` 时时间片流转，虽然也有优先级的数字，但是即使是最高优先级的进程，在时间片用完的时候也会释放 CPU。而 `SCHED_FIFO` 除非是主动释放，



否则具有最高优先级的进程永远不会释放 CPU（等待 IO 完成等阻塞操作除外）。当两者存在更高优先级进程时都会被抢占。

前面说了没有实现 `SCHED_IDLE` 调度的方式，那么 Linux 如何实现后台磁盘整理等操作呢？答案是使用功能类似的 `SCHED_BATCH` 调度方式。这种调度方式并不会在有正常程序的时候完全不执行，但是会保证正常程序的执行和交互程序的响应，也适合 GCC 等编译操作。

### 5.2.2 进程调度策略的配置

可以通过内核提供的 API 设置调度的办法，也可以通过命令行设置进程调度，命令是 `chrt`，还可以配置实时进程的最大时间占用，因为如果实时进程出现 Bug，那么最高优先级的进程几乎不可能释放 CPU，将导致系统卡死。通过 `sysctl` 调用可以设置 `kernel.sched_rt_period_us` 等参数，可以配置最大的实时调度进程占用的 CPU。

通过搭配 `cgroup` 和进程调度，还可以实现按照 `cgroup` 进行 CPU 资源的配置方式，这也是通过 `cgroup` 文件系统完成的。

### 5.2.3 公平问题

绝对的公平是不存在的，社会上人与人之间能获得的社会资源和自然资源也是不一样的。进程也是如此，因此所有的进程调度算法都会有优先级的划分。我们都知道现在的大部分进程的调度算法都是划分 CPU 的时间片，但是以前 UNIX 划分绝对的时间片的方法已经被 Linux 淘汰了，Linux 改用基于比例的划分模型，而不是基于绝对时间片的方法。例如如果是绝对时间不同的同优先级的进程分别获得 10ms 和 5ms 的时间片，相差了一个时间片（假设一个是 5ms），且将 90ms 和 85ms 两个划分也只相差了一个 5ms 的时间片，但是这在应用程序的表现上就有极大的区别，因为前者是两倍的差距，后者差距程度相比自身的时间片来说微不足道。前者由于时间片本身就短，还得担负更多进程切换的开销，所以时间片就更短了。因此绝对时间是对机器友好，而不是对用户友好，但使用这个算法的是用户。

Linux 的做法是使用 CFS，即使用完全公平的方法来实现不公平。简单地说，就是根据目前有多少个进程，用总 CPU 资源除以进程个数就知道每个进程要占多久

的时间片了。如果要定义有高优先级的进程，可以让它占有多份时间（但不一定是整数）。比起绝对时间的最大区别是根据进程数目的比例时间。例如优先级是 5 和 0 的两个进程与优先级是 10 和 15 的两个进程，在一个 CPU 上获得的时间片是一样的（假设只有这两个进程），因为使用的是差值和比例计算的（具体算法略过）。学过 `cgroup` 的读者就会发现，这种被叫作完全公平调度的算法是整个系统不公平调度的基础，`cgroup` 就是使用公平的 CFS 划分出一部分占比，然后在占比内使用 CFS，如此 `cgroup` 内的公平相对于 `cgroup` 外部就是完全不公平的，因为这个 `cgroup` 子系统占据的比例本身就是不同的。

## 5.2.4 内核线程的调度

内核中的很多操作都是使用一些内核基础设施完成的，例如 `workqueue`、`tasklet`、`softirq` 等，这些基础设施一般可以完成特定的任务。既然是用来完成任务的，就必须参与调度，而调度的单位只能是内核线程。所以这些机制虽然对用户来说是一些拿来即用的调用接口，但其执行却是通过特定的内核守护线程执行的。

Linux 中的中断分为上下两部分，下半部分可以关中断，产生上半部分的中断任务；上半部分不需要关中断，可以调度执行。出现这种情况的原因是系统中关中断的时间必须短，否则就会失去响应。产生的软中断被加入内核守护线程 `ksoftirqd` 的执行队列，这个线程后续会调度执行相关软中断。`tasklet` 与软中断类似，只是在 SMP 系统中，软中断可以被多个 CPU 一起执行，是可重入的。而 `tasklet` 一次只允许一个 CPU 执行，是不可重入的。用户可以根据软中断是否允许重入来决定是否使用 `tasklet` 或 `softirq`。

因为 `softirq` 和 `tasklet` 不能睡眠，所以不能使用信号量或其他阻塞函数，而且它们对应的都是由一个内核线程执行的（`ksoftirqd`），如果阻塞了，系统将无法响应其他软中断。工作队列 `workqueue` 本身就是作为一个可用的单元提供给用户的，一个 `workqueue` 就是一个内核线程（前提是不使用 `pool` 的 `kworker` 内核线程）。内核模块可以生成一个 `workqueue`，然后添加自己的任务进去，也可以使用内核已有的 `workqueue`，向其中添加任务。`workqueue` 是一个容器，内核模块可以向已有的 `workqueue` 中添加任务。该 `workqueue` 就会调度执行自己的子任务，可以说是进程中的进程。

## 5.3 资源

### 5.3.1 资源锁

内核中的资源锁有自旋锁、信号量、互斥锁、读写锁（rwlock）、顺序锁、RCU 锁和 futex 锁。这些锁分别用来解决不同类型的问题，具体如下。

- 软中断中多个 CPU 同时访问同一资源。由于软中断不能睡眠，因此在多个 CPU 抢用同一个资源时不能使用其他锁，只能忙等，这就是自旋锁。
- 在普通进程竞争资源时，该资源无论是读还是写，在同一时间只能有一个或几个进程获得。这就是互斥锁和信号量（信号量为 1 时就是互斥锁）。
- 当互斥不是很频繁的时候，希望不必每次都进入内核，就使用 futex 锁。
- 同一个资源希望读和写分开处理，就使用读写锁、顺序锁和 RCU 锁。

不同的锁服务于不同的目的和场景。实际上 Linux 只是应用资源锁思想的一部分，操作系统有多种方式用于处理资源锁的问题。

资源锁在本质上是同步和互斥的问题。从上面的内容可以看出，大部分是处理同时写的问题。所以只要能保证比较和写的操作都是原子的，线程就可以是无锁的。Intel 已经实现了类似的指令，比如 cmpxchg8，在一个周期内完成比较和写操作就可以保证不发生并发写冲突。

同样的思想，Linux 也提供了两组原子操作：一组针对整数；另一组针对位。合理地利用原子操作就可以避免大部分的锁应用场景。使用自旋锁代价很大，一个 CPU 运行时需要另外的 CPU 空转等待，但是当要锁住的代码量很少时，由于自旋锁的轻量级比使用信号量付出的代价小很多。所以，自旋锁不仅用于软中断，还可以用于给很少的一段代码加锁，示例如下。

```
typedef struct {
    raw_spinlock_t raw_lock;
} spinlock_t;
typedef struct {
    volatile unsigned int slock;
} raw_spinlock_t;
spin_lock_init();    //把自旋锁设置为 1，未锁的状态
spin_lock();         //循环，直到自旋锁变为 1，然后把自旋锁设置为 0
```



```
spin_unlock();           //把自旋锁设置为 1
spin_unlock_wait();      //等待, 直到自旋锁变为 1
spin_is_locked();        //如果自旋锁设置为 1, 返回 0, 否则返回 1
spin_trylock();          //把自旋锁设置为 0, 若原来锁是 1, 则返回 1, 否则返回 0
```

以上简单的模型, 可以看出自旋锁的核心原理就是一个取值为 0 或 1 的整数的加減问题。自旋锁只在 SMP 中才有意义, 否则一个 CPU 自旋就会永久堵塞。当一个自旋锁上有很多进程在自旋等待时, 就可以判断在自旋锁上的操作非常忙。判断的方式是在自旋的过程中会发现自旋锁的所有者发生了改变, 此时, 应该睡眠而不是继续自旋。

除了自旋锁, 还有一种锁需要忙等, 这就是顺序锁。严格地说这不是忙等, 而是使用了一个巧妙而又非常简单的思想: 在读之前看锁值, 在读之后看锁值, 如果不变化, 就表明在读的过程中, 读的值没有被写, 就不需要重读, 否则就重读。写的时候就会改变锁值, 原理相当于自旋锁, 但是可以允许多个写。读操作在多个写操作全部完成后才能读得正确的值。

当要加锁的是复杂的逻辑时, 就需要信号量这种重量级的锁, 但是一般的逻辑都应该尽量避免大块代码的加锁。在实际操作中, 也可以通过精细的设计来避免大块锁。信号量有一个问题, 如果多个 CPU 获得读锁, 则信号量本身会在各个 CPU 的 cache 中不断地刷新, 造成效率降低。解决的方式是内核定义了一个新型的信号量: percpu-rw-semaphore。

RCU 锁不阻塞写, 前面的顺序锁已经是改进的读写锁了, 但同时也只能有一个写操作。但 RCU 锁允许不阻塞写操作, 进行多个写操作时不是写到同一个地方, 而是拷贝一份新的数据进行写操作。读操作还继续读旧的, 这样以增多内存的使用为代价换来读写都不阻塞。

还有一种仅由用户空间进程使用的锁——futex。使用这个锁可以完全取代用户空间的各种锁 (用户态很多高级语言的锁是封装的 futex), 因为其高效, 行为又符合要求。futex 的原理其实是考虑到用户使用的信号量等锁都是内核中的一个变量, 每次查询的时候都要进入内核态, 然后再出来。futex 的思想就是直接将内核态的这个锁变量 mmap 映射到用户进程空间中, 这样各个用户进程就可以在自己的空间直接查询这个值, 而不用进入内核就可以知道有没有人在使用。虽然读取是大家都可以随便读, 但是写入考虑到多个进程操作一个变量可能发生冲突, Linux 是提供 API 陷入内核来加锁写入的。虽然最后还是要陷入内核, 但是其判断部分可以不进入内

核完成，大部分进入的情况判断资源是没有并发访问的，特殊应用场景除外。

`futex` 是用户端使用锁的一个很好的选择，然而用户的进程具有不同的优先级，而锁无视所有优先级，信号量可以实现同步概念，但是锁没有。有些时候希望获得锁在进程上有优先级的区别，这是 `pi-futex` 锁提供的功能，叫作优先级继承，是使用 `futex` 锁实现的，增加了判断进程优先级来确定解锁的优先级。打开这个机能之后整体效率会显著下降，个别进程会获得更多的效率。

资源被抢占的情况有两种：`SMP` 系统下多个 `CPU` 的并发访问和一个 `CPU` 下的可抢占访问。大部分应用在开发时都使用一样的锁来锁数据，这两种情况有不一样的特点。在很多情况下，一个 `CPU` 的可抢占锁可以做得更轻巧。

通过 `preempt_enable()`、`preempt_disable()`、`preempt_enable_no_resched()`、`preempt_count()`、`preempt_check_resched()` 这几个函数可以在可抢占单 `CPU` 的情况下完成锁的工作，就不需要其他种类的锁了。主要应用在中断处理中，禁止流程被中断抢占，示例如下。

```
preempt_disable();
load %r0,num
add %r0,1 //发生中断
store %r,0 num
preempt_enable();
```

这样就不能在中间代码发生中断，也避免了在 `load` 之后中断可能修改 `num` 大小的这种情况的发生。

最有效的资源访问方式就是不加锁，大部分的问题都可以使用无锁设计来实现，但是这样可能带来内存上更大的损耗和代码更难管理的问题。

### 5.3.2 资源限制

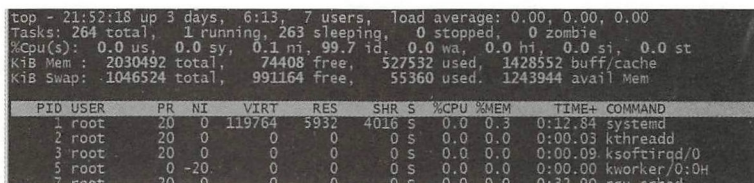
互斥概念与同步概念必须要区别开。互斥是指同一时间只有一个进程可以访问资源，没有时序概念，而同步包含了多个访问该资源进程的访问的先后顺序，有“你结束了轮到我了”的意思。互斥的意思是“你还没结束我就没法开始”。信号量是同步概念的，因为未得到资源的进程会睡眠等待。其他的内核锁是互斥概念的（自旋、顺序），因为得不到就阻塞，或者是让其永远可以得到（`RCU`）。

资源限制常用 `cgroup`，随着 `Docker` 的成熟，`Docker` 被更广泛地使用。在这之前，可以独立使用 `getrlimit` 和 `setrlimit` 这两个系统调用进行资源限制。如果你在做一個可发布的应用程序，`rlimit` 仍是一个不错的选择。`cgroup` 可以限制 CPU、内存、文件、行为等，甚至系统调用。限制进程的可见的系统调用使用 `seccomp_filter` 功能。

### 5.3.3 进程对系统内存的使用

大部分进程通过 `glibc` 申请使用内存，但是 `glibc` 也是一个应用程序库，它最终也是要调用操作系统的内存管理接口来使用内存。在大部分情况下，`glibc` 对用户和操作系统是透明的，所以直接观察操作系统记录的进程对内存的使用情况有很大的帮助。但是 `glibc` 自己的实现也是有问题的，所以在特殊情况下追究进程的内存使用也要考虑 `glibc` 的因素（例如很多 `glibc` 版本在处理大量并发小内存的时候会出现内存泄漏）。其他操作系统资源使用情况则可以直接通过 `proc` 文件系统查看。

进程需要内存，但并不一定需要物理内存。一个进程可以申请 1GB 内存，内核也确实会批准给它（取决于 `/proc/sys/vm/overcommit_*`），但是内核在真正给它安排实际对应的物理内存的时候是在进程需要实际使用内存的时候，这个时候会引发缺页异常，内核就在这个异常处理代码时把实际的内存安排给进程。就像你在银行存钱时，很多时候你的资产都只是一个“数目”，只有当你要取出现金的时候银行才有必要筹措现金支付给你（因为它之前承诺过），但是银行的现金数永远小于储户的总资产数。当大家都要兑现时，银行可能会破产。操作系统的内存也一样，当进程都要求兑现的时候，内核不一定能够全部兑现，而且会崩溃。崩溃的时候会根据当前每个进程的 OOM 分数选择当前分数最高的进程直接“杀死”（关于 OOM 的详细算法与机制这里不讨论），如果你的进程过度占用内存而被杀死，则可以在 `/var/log/kern.log` 查看相关信息，如图 5-1 所示。



```
top - 21:52:18 up 3 days, 6:13, 7 users, load average: 0.00, 0.00, 0.00
Tasks: 264 total, 1 running, 263 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.1 ni, 99.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 2030492 total, 74408 free, 527532 used, 1428552 buff/cache
KiB Swap: 1046524 total, 991164 free, 55360 used, 1243944 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1	root	20	0	119764	5932	4016	S	0.0	0.3	0:12.84	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.03	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:00.09	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	20	0	0	0	0	S	0.0	0.0	0:32.90	rcu_sched

图 5-1



进程的内存种类有：用来存放数据的堆；用来执行进程的栈；与其他进程的物理内存（SHR，例如共享库，共享内存）；进程的虚拟地址空间大小（VIRT）；应用进程实际正在使用的物理地址的大小（RSS）；进程用来放要执行的代码的部分。总体来说，应用程序实际要使用的物理地址由数据、栈、可执行代码存放这三部分组成。一般大部分进程需要最多的是数据内存，因为数据内存的需求是可以大幅度变化的。

查看一个进程资源的最好方法是使用 `/proc/pid/stat*` 中的这三个文件：`statm`、`status`、`stat`。其中 `statm` 是关于进程的内存使用信息的；`stat` 是非常全面的信息，例如进程号、缺页次数、启动时间、信号、CPU、进程组等；而 `status` 是一个可读的、用户通常比较关心的内容。另外还有一个 `smaps`，在这个文件中可以看到非常详细的进程内部内存的映射情况，而 `maps` 文件则是一个概览。

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：`brk` 和 `mmap`（不考虑共享内存）。

(1) `brk` 是将数据段（`.data`）的最高地址指针 `_edata` 往高地址推；

(2) `mmap` 是在进程的虚拟地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。

这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系。在标准 C 库中，提供了 `malloc`、`free` 函数分配释放内存，这两个函数底层是由 `brk`、`mmap`、`munmap` 这些系统调用综合实现的，并不是每一次 `malloc`、`glibc` 都会到内核去使用 `brk` 申请内存，而是 `glibc` 申请了一大块内存，然后组织分发给对应的进程，只有不够用了才会再次向内核申请。

## 5.4 多进程与进程通信

### 5.4.1 多进程模型

在内核看来，每一个进程能访问的资源通常是其他进程不知道的，而用户态编程要求多线程或多进程编程可以共享部分内核数据，Linux 内核中解决这个问题的方

式是使用一个 `clone` 机制，使得一个进程在创建时可以指定哪些资源可以与其他进程共享。从而模拟实现多线程环境。较新的内核不但可以共享资源，还可以使用 `unshare` 系统调用取消共享。

我们知道进程是被制造出来的概念，那么 Linux 是如何制造这个概念的呢？前面说了调度和资源竞争的解决方法，那么调度的究竟是什么呢？

例如我们自己写一个机场飞机起飞的调度程序，根据一系列的原因（跑道资源、天气原因、上级安排等）安排不同飞机在不同的跑道上起飞，我们安排的是飞机，那么在程序中我们如何表示飞机呢？那必然是一个结构体（在 C++ 中可以是一个类）。这也就不难理解进程调度算法调度的是怎么了，其实也是一个结构体，这个结构体是 `task_struct`，一个非常大的结构体。进入调度算法，当调度算法运行结束后，其必然输出的是一个 `task_struct` 结构体（`current` 宏）。而 CPU 执行的永远是调度算法执行结束后的输出结构体所描述的代码位置。

我们知道任何现代程序的执行都要有栈的概念，栈最大的功能是用来做函数跳转（其实又是一个为了达到函数目的的代价产品）。而栈有大小；有组织结构；有当前的位置；有定义好的出栈、入栈的操作方法。在没有进程概念时，只需要一个栈，就是内核代码运行的栈，而有了进程概念之后，就要为每一个进程准备单独的栈了，而这个工作只能由内核自己来完成。

为了实现进程概念，带来的又何止是栈设计与维护这点工作量呢？如何有效地定位各个 `task_struct` 呢？自然是靠数字，于是有了 `pid` 概念。在进程被调度算法切换出 CPU 时，进程执行时存储在寄存器上的变量怎么办？只能设计机制来保存与恢复，于是有了进程上下文的概念。进程作为一个实体与其他进程的关系应该怎么定义？于是有了进程家族树的概念。于是进程如何创建、如何终结，又带来了很多新的概念。这一切的代价，都得由内核去弥补。

确切地说，介绍内核的进程就是介绍内核如何处理进程概念的引入带来的一系列代价，而至于进程概念本身，在所有操作系统都是一样的，因为它只是一个存在于理论上的概念模型。

## 5.4.2 用户进程间通信

应用程序之间的通信有很多种方法，比如使用前面重点介绍过的 FIFO 文件，

mmap 映射一块共享内存, System V IPC 或 POSIX IPC 中的三种常用方式(信号量、共享内存、消息队列), 匿名的 pipe 管道。可以是大型应用常用的 UNIX Domain Socket 或者普通 socket (甚至可以是 TIPC socket)。还可以使用一些不常用的, 比如非常强大的 mailbox 机制、用户端服务的 DBUS, 以及很少有人会听说的 NETLINK\_USERSOCK。

DBUS 曾经被 Fedora 系列重度使用, 其不是直接使用内核的机制, 而是在用户端建立了通信管道, 以至于早期开发的很多软件都使用 DBUS 机制。但是随着 Linux 的发展和 Fedora 的没落, DBUS 也在逐渐被抛弃。由于 FIFO 文件使用复杂, 并且较老, 除非认真封装, 否则在大部分情况下只适合简单命令行, 所以在实际的工程产品中使用较少。匿名管道多用于父进程启动一个子进程, 并且与子进程之间通信, 前面的内容也重点介绍过, 并且给出了使用方法。mailbox 在内核的 3.18 版本才出现, 可以模拟现实的邮箱应用。一个进程能给所有其他进程发送邮件, 但是只有本进程可以接收发送给自己的邮件。每个进程只有一个邮箱地址, 邮件的处理顺序是 FIFO (先进先出)。目前没有太多的知名服务采用 mailbox 机制, 关于 mailbox 机制的相关例子和文档也较少, 目前稳定性也没有太多考量。使用 mmap 做内存映射比较适合安全领域应用和特别高级的用户, 并且其能达到的效果也类似于 system v IPC 的共享内存。所以此种方案一般也不直接用于进程间通信的可选方法。NETLINK\_USERSOCK 是一个比较奇特的通信方法, 使用 netlink 拥有 socket 的通信方式, 又有 mailbox 的通信能力, 理论上应当非常强大, 但是很少有人在实际操作中使用。目前如果要进行用户进程通信, 最好的选择仍旧是 System v IPC (POSIX IPC) 的三种方法, 大量数据通信的很多服务也喜欢采用 UNIX Domain Socket, 而使用 loopback 环回设备的 socket 通信效率不如 UNIX Domain Socket, 所以一般不用, 因为 loopback 设备需要处理大量的 tcp 逻辑, 而大部分的 tcp 逻辑是相对低效的。

在 system v ipc 中, 信号量一般被用于资源限制, 由于共享内存需要手动地组织内存的安排, 所以有相当高的复杂度。所以更常见的进程间通信的方案是消息队列, 其非常方便和容易组织, 示例如下。

```
void init_channel(){
    try{
        message_queue::remove(CHANNEL_NAME);
        message_queue mq(create_only, //only create
                           CHANNEL_NAME, //name
                           1000, //max message number
```



```

        sizeof(Message) //max message size
    );
} catch(interprocess_exception &ex){
    BOOST_LOG_SEV(slg,Common::normal) << ex.what() <<
std::endl;
    BOOST_LOG_SEV(slg,Common::normal)<<"message queue create
failed";
}
std::thread(message_listen).detach();
}

```

如上一个 C++函数，即可完成消息队列的创建，通常创建之后还需要一个单独的线程进行消息的监听和处理监听逻辑。C 语言的接口代码也非常简单，注意队列的名字前要加“/”。下面是 message\_listen 线程，示例如下。

```

struct Message{
    uint64_t id;
    uint64_t cost;
};
void message_listen(){
    try{
        message_queue mq(open_only,CHANNEL_NAME);
        unsigned int priority;
        size_t recvd_size;
        Message msg;
        while(true){
            mq.receive(&msg, sizeof(msg), recvd_size,
priority);
            if(recvd_size != sizeof(msg)){
                BOOST_LOG_SEV(slg,Common::normal)<<"message receive error";
                continue;
            }
            //进行 message 消息处理
            rulestats[msg.id].exec_time_sum += msg.cost;
            rulestats[msg.id].exec_num++;
            rulestats[msg.id].id = msg.id;
        }
    }
}

```

```

    }
    catch(interprocess_exception &ex){
        BOOST_LOG_SEV(slg,Common::normal) << ex.what() << std::endl;
    }
}

```

Listen 线程循环阻塞监听，监听到的 message 可以直接解析已经定义好的结构体，由于结构体可以与程序代码无缝整合，所以在编程上也带来了极大的方便。ipcs 命令可以用来查看当前的消息队列。也可以使用 mkdir /dev/mqueue 和 mount -t mqueue none /dev/mqueue 两个命令进行挂载。挂载之后就可以当前的消息队列文件了。现代系统会默认挂在到/run/shm/目录，使用 ls 命令就可以看到，如图 5-2 所示。

```

root@ubuntu:/# ls /dev/mqueue/
test_queue

```

图 5-2

消息队列并不一定能创建成功，在/proc/sys/fs/mqueue/下有三个文件是用来限制消息队列的参数的，如果超过了这个参数，在创建消息队列时就会有 Invalid argument 的错误。此外 rlimit 也可以用于限制队列。发送消息的示例如下。

```

try{
    message_queue mq(open_only,CHANNEL_NAME);
    Message msg={id, cost};
    if(mq.try_send( &msg, sizeof(msg), 1) == false){
        BOOST_LOG_SEV(slg,Common::normal)<<"send to message queue
failed,rule id:"<<id<<endl;
    }
}

catch(interprocess_exception &ex){
    BOOST_LOG_SEV(slg,Common::normal) << ex.what() << std::endl;
    BOOST_LOG_SEV(slg,Common::normal) <<"send to mssage queue
failed"<<endl;
}

```

以上的代码是其他进程想要发送消息给监听进程的全部过程。可以看到，只需要打开同名的消息队列，然后 send 即可。在这里即使是直接使用 C 语言的底层接口也不会太难。



至于 UNIX Domain Socket 的使用,则与普通 socket 的使用没有太大差别,很多人也正是因为这样才会选择使用这个方式。

### 5.4.3 内核与用户空间的进程通信

#### 1. Netlink

Netlink 是用户程序与内核通信的 socket 方法,通过 Netlink 可以获得修改内核的配置。常见的例如获得接口的 IP 地址列表、更改路由表或邻居表等。老版本的内核提供很多其他方式从内核获取信息,至今仍在被广泛使用。Netlink 作为一种通信方式在 Linux 系统中的地位越来越高,因此这里会详细介绍。

#### 2. Netlink 消息格式

Netlink 使用通用的 socket 接口,只是添加了一个新的类型。创建 Netlink 的 socket 的方法,代码如下。

```
#include<asm/types.h>
#include <sys/socket.h>
#include <linux/netlink.h>
netlink_socket =socket(AF_NETLINK, socket_type, netlink_family);
```

socket\_type 只可以是 SOCK\_RAW 或者 SOCK\_DGRAM,内核并不区分这两种类型,所以用户使用哪个都可以。而 netlink\_family 就是用来选择具体的 Netlink 在内核端沟通的模块,最常用的是 NETLINK\_ROUTE,IP 地址、路由表、邻居表等都是在这个 family 中。其他的例如 NETLINK\_SOCK\_DIAG 可以用来查看详细的 socket 信息。代码如下。

```
//Netlink 的请求头部结构体:
struct nlmsghdr{
    __u32 nlmsg_len;
    __u16 nlmsg_type;
    __u16 nlmsg_flags;
    __u32 nlmsg_seq;
    __u32 nlmsg_pid;
};
```



任何一个具体的消息在这个 struct nlmsgghdr 之后都要紧跟具体功能对应的结构体（例如 inet\_diag 的请求就需要紧跟 inet\_diag 的头部），所以这里有个 nlmsg\_len 域，用来表示 Netlink 头部加上后面具体功能的请求头部一起的长度。

nlmsg\_type 是后端对应的功能模块，随着内核功能的完善，这个支持的模块也在增长。nlmsg\_flags 就是针对操作的标志。例如一个 Netlink 请求的头部允许有多个 nlmsgghdr，如果是这样，每一个 nlmsg\_flags 域都要在 flags 域设置 NLM\_F\_MULTI，最后一个 flags 设置 NLMSG\_DONE。多个 nlmsgghdr 结构体的情况，每个头部的数据都紧跟在这个头部的后面。

nlmsg\_pid 用来表示发送这个请求的进程 pid（所以你可以伪造为其他进程发送），nlmsg\_seq 是用户自己设置的，内核的返回也会回复这个，可以让用户用来追踪任何一个请求，经常会使用时间戳作为这个值。如果嫌麻烦，可以在 bind 的时候填好 pid，这里可以直接设置为 0，没有任何问题。内核回复的时候也是一个 struct nlmsgghdr 通用头部后面跟具体的功能头部，例如 inet\_diag，再后面是具体功能对应的数据。例如请求 ip 列表，就返回 ip 列表数据。操作返回数据有很多宏定义，例如 NLMSG\_PAYLOAD、NLMSG\_DATA 等，建议阅读宏的代码，清楚内部实际的操作。

Netlink 消息格式，如图 5-3 所示。可以看到首先是 nlmsgghdr，然后是 payload。而 payload 又可以进一步划分，先是 Netlink、family 等的头部，然后是属性列表，中间都是有 pad 的，这个 pad 是 4 字节对齐的。Netlink 消息属性格式如图 5-4 所示，代码如下。

```

*                                     Messages Interface
*
* -----
*
* Message Format:
* <--- nlmsg_total_size(payload) --->
* <-- nlmsg_msg_size(payload) -->
*
* +-----+-----+-----+-----+-----+
* | nlmsgghdr | Pad | Payload | Pad | nlmsgghdr |
* +-----+-----+-----+-----+-----+
* nlmsg_data(nlh)---^                                     ^
* nlmsg_next(nlh)-----+
*
* Payload Format:
* <----- nlmsg_len(nlh) ----->
* <----- hdrlen -----> <- nlmsg_attrlen(nlh, hdrlen) ->
*
* +-----+-----+-----+-----+-----+
* | Family Header | Pad | Attributes |
* +-----+-----+-----+-----+-----+
* nlmsg_attrdata(nlh, hdrlen)---^

```

图 5-3

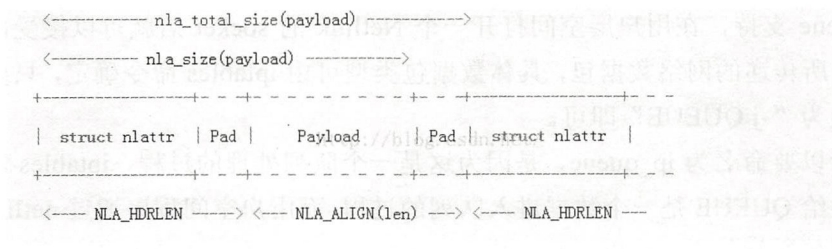


图 5-4

```
struct nlattr {
    __u16      nla_len;
    __u16      nla_type;
};
```

例如 genetlink 的 family 头部就是 `struct genlmsghdr`，然后再是用户自己的头部，最后是属性列表。

#### 5.4.4 Netlink 功能模块

NETLINK\_ROUTE 用来与邻居表、路由表、数据包分类器、网卡信息等路由子系统通信，获取信息或者设置。这个为用户端最常见的类型，netfilter 项目基于 libnetlink 的 libmnl 库供上层应用直接使用，就不用手动封装复杂的 Netlink 系统了。

NETLINK\_W1 就是 GPIO 用来拉高或者拉低某一根线的内核子系统，所以用户如果使用 GPIO 就可以不用动内核，直接在用户空间操作 GPIO 就可以了（树莓派使用 dev 下的 GPIO 设备，还有强大的 Python GPIO 库）。

NETLINK\_USERSOCK 就是用户端 socket，使用这个处理 Netlink 请求的单位就不是内核了，而是用户空间的另外一头的某个进程。这个就是进程间通信的另一种方案。由于是 socket，一端可以监听，另一端只要将发送的目标地址填充为目标进程的 pid 就好（Netlink 的发送地址不是 ip 编码的，而是 pid 等编码的）。这种 IPC 最厉害的地方在于可以支持 multicast，多播的通信。一个消息可以同时发送给多个接收者，但是普通的回环地址 lo 的 socket 通信也可以做到这一点。

NETLINK\_FIREWALL 是跟内核的 netfilter 的 ip\_queue 模块沟通的选项。ip\_queue 是 netfilter 提供的将网络数据包从内核传递到用户空间的方法，内核中要提

供 `ip_queue` 支持，在用户层空间打开一个 Netlink 的 socket 后就可以接受内核通过 `ip_queue` 所传递的网络数据包，具体数据包类型可由 `iptables` 命令确定，只要将规则动作设置为 “`-j QUEUE`” 即可。

之所以要命名为 `ip_queue`，是因为这是一个队列处理的过程，`iptables` 规则把指定的包发给 `QUEUE` 是一个数据进入队列的过程，而用户空间程序通过 `netlink socket` 获取数据包进行裁定，结果返回内核，进行出队列的操作。在 `iptables` 代码中，提供了 `libipq` 库，封装了对 `ipq` 的一些操作，用户层程序可以直接使用 `libipq` 库函数处理数据。`NETLINK_IP6_FW` 与 `NETLINK_FIREWALL` 的功能一样，只是专门针对 IPv6 操作。

`NETLINK_INET_DIAG` 就是同网络诊断模块通信使用的，最常用的是 `tcp_diag` 模块，可以获得 TCP 连接的最详细信息。

- `NETLINK_NFLOG` 是内核用来将 `netfilter` 的日志发送到用户空间的方法。
- `NETLINK_XFRM` 是与内核的 `ipsec` 子模块通信的机制。

`NETLINK_FIB_LOOKUP` 用户可以自由查询 FIB (Forwarding Information Base) 路由表。FIB 是快速转发表，里面数据量很大，刷新比较快，服务于快速查找和快速转发，而不是服务于用户空间设置的。用户空间设置使用的路由表是 RIB (Routing Information Base)，在内核中 RIB 会转化为 FIB。也就是说 RIB 是控制平面的，FIB 是数据平面的。`NETLINK_NETFILTER` 用于控制 `netfilter`。

## 5.4.5 其他 Netlink 种类

其他 Netlink 种类如下。

- `NETLINK_SELINUX` 与内核的 `selinux` 通信。
- `NETLINK_ISCSI` 是 `open iscsi` 的内核部分，通过 `iscsi` 可以组成 `iscsi` 网络，服务于网路存储系统。
- `NETLINK_AUDIT` 与内核的 `audit` 模块通信，记录了一大堆的事件。
- `NETLINK_CONNECTOR` 是内核端的模块，如果想要使用 Netlink 接口对用户提供服务，可以用这个模块注册一个 Netlink 回调，用户空间使用这个子系统就可以连接到特定的内核模块了。
- `NETLINK_KOBJECT_UEVENT`: `sys` 子系统使用的 `uevent` 事件。内核内所有设



备的 uevent 事件都会通过这个接口发送到用户空间中。早期的 Linux 使用 hotplug 机制，可以在 `/proc/sys/kernel/hotplug` 文件中指定当有设备插入时调用的程序，至今仍被广泛使用在嵌入式系统中。

- **NETLINK\_GENERIC**: 内核模块用来提供 Netlink 接口的方式。通过这种方式提供的接口都可以复用这一个通用子系统。
- **NETLINK\_CRYPTO**: 可以使用内核的加密系统或者修改查询内核的加密系统参数。

### 1. Netlink 请求 `nlmsg_flags`

由于涉及具体的后端请求类型，所以在设计这个 flag 时尽可能设计成通用的，在不同的后端会有不同的表现。大家可以大概了解一下每一个 flag 的意思，但是在使用时要根据不同的用途区别对待。

### 2. 与具体模块无关的通用设置

- **NLM\_F\_REQUEST**: 所有请求类型的 Netlink 都会设置。
- **NLM\_F\_MULTI**: 用于表示多个 Netlink 请求在同一个包的 `NLMSG_DONE` 结尾的最后一个头部。
- **NLM\_F\_ACK**: 由于 Netlink 是不可靠的，可以通过让内核回复 ACK 模拟的实现可靠。其实在绝大多数情况下是可靠的，如果不可靠则说明内存不够了。如果是查询信息，例如 `RTM_GETADDR` 是用于获得 ip 地址的请求类型，内核会返回每个接口的信息。此时再要求返回 ACK 就显得多余了。
- **NLM\_F\_ECHO**: 这是让内核响应这个请求，一般需要内核响应（但也不是所有的内核子系统都是按照这个模型设计的）。如果不设置，很可能只有 ACK（如果设置了 `NLM_F_ACK`）。

### 3. 专门为 GET 类的请求附带的 flag

- **NLM\_F\_ROOT**: 返回满足条件的整个表，而不是单个的 entry。
- **NLM\_F\_MATCH**: 返回所有匹配，这个在内核中只是提供了一个接口，并没有具体的实现，所以是否设置不设置都无所谓。但是比如 `tcp_diag` 根据 `sockid` 获取单条 TCP 连接信息的功能就可以使用这个标志，只是目前还没有实现而已。
- **NLM\_F\_ATOMIC**: 请求返回表的时候返回的是一个快照。
- **NLM\_F\_DUMP**: 这个是 `NLM_F_ROOT` 和 `NLM_F_MATCH` 的组合，意思是返回全部满足指定条件的条目。比如现在设置 `RTM_GETADDR`，请求返回 interface

index 为 1 的结果，但是由于内核没有实现 NLM\_F\_MATCH，所以只要你使用了 NLM\_F\_ROOT，无论如何设置 index 值，结果都是全部返回。

#### 4. 专为 SET 类的请求附带的 flag

- NLM\_F\_REPLACE：取代已经存在的匹配条目。
- NLM\_F\_EXCL：如果条目已经存在就不取代。
- NLM\_F\_CREATE：如果不存在就创建。
- NLM\_F\_APPEND：加在对象列表的最后。

以上的 flag 组合设置有常用的设置模式，但是具体的情况与每一个模块的意义是相同的。Netlink 的架构设计不是很自然，因为在 Netlink 的通用头部指定了操作的类型，而具体的操作已经代表了操作的类型，这相当于指定了两遍。也就是使用具体功能的人不但需要知道具体的功能函数，还要知道这个功能函数对应该如何设置 Netlink 的 flag。一个典型的请求类的 flags 的设置方法如下。

```
req.hdr.nlmsg_flags = NLM_F_DUMP | NLM_F_REQUEST;
```

#### 5. Netlink 的请求类型 nlmsg\_type

这个与具体的后端模块相关的，不是 Netlink，但还是提供了集中通用的消息类型。例如 inet\_diag 定义了 TCPDIAG\_GETSOCK 和 DCCPDIAG\_GETSOCK 这两种类型的 type；在 include/uapi/rtnetlink.h 中为定义 rtnetlink 定义了很多 type，这些 type 被称作 action 会更加贴切一些。代码如下。

```
req.hdr.nlmsg_type = RTM_GETADDR;
```

#### 6. Netlink 策略：nla\_policy

这个 Netlink 被其他模块用到的情况比较少，在 genetlink 中有提供使用的接口，但是 genetlink 的使用者仍然可以选择不使用它。

### 5.4.6 genetlink 的使用

#### 1. 请求流程

前面说过 Netlink 有一种通用的用法是 NETLINK\_GENERIC 类型，可以供大家自由地扩展使用 Netlink 功能。内核模块开发者可以使用这个类型直接提供 Netlink 能

力（完整的代码可以参考网址：<https://github.com/archerbroler/tcpinfo/tree/master/v3>），并且建议读者边写测试程序边阅读实例程序。

这一类的 Netlink 内核有专门的头部定义，用户发来的 Netlink 消息，先是 Netlink 头部，然后紧跟 struct genlmsg\_hdr 这个 genetlink 头部，最后在这个头部之后紧跟用户自己定义的头部，代码如下。

```
struct genlmsg_hdr {
    __u8      cmd;
    __u8      version;
    __u16     reserved;
};
```

当这个消息传递到内核的时候，genetlink 的内核部分会自动解析，将这个头部对应的内核解析，代码如下。

```
struct genl_info {
    u32          snd_seq;
    u32          snd_portid;
    struct nlmsg_hdr * nlhdr;
    struct genlmsg_hdr * genlhdr;
    void*        userhdr;
    struct nlattr ** attrs;
    possible_net_t _net;
    void*        user_ptr[2];
    struct sock * dst_sk;
};
```

这个结构体交给使用 genetlink 架构处理 genetlink 消息的内核模块使用。这些都是从用户提交的请求中解析出来，形成的 struct genl\_info 结构体，并且只会传递给 do\_it 调用。

nlhdr、genlhdr、userhdr 分别是 Netlink 头部、genetlink 头部、用户自定义头部，attrs 是 Netlink 的属性机制。如果检测到消息中有属性，程序就会解析，然后放到这个位置传递给实际的处理函数进行处理（可以使用属性也可以直接用用户自定义的请求头部，官方推荐使用属性，认为这样可以增加可扩展性，方便维护）。



## 2. 概念概览

genetlink 既然是提供给多个内核模块使用的，就一定会提供区分不同内核模块的方法机制。genetlink 通过 family 和 ops 两个维度来定位到最终的处理函数。

## 3. family

family 一般用来确定模块。要使用 genetlink 的内核模块首先定义一个 family，代码如下。

```
static struct genl_family my_gnl_family = {
    .id= NETLINK_FAMILY_ID,
    .hdrsize = 0,
    .name = "my_netlink",
    .version = 1,
    .maxattr = A_MAX,
};
```

其中 id 是用户自己指定的数字，用户端在填充 Netlink 的请求头部的时候要填写这个 nlmsg\_type 作为 Netlink 请求的目标类型。这个指定的数字不要与已有的数字相冲突，否则内核就无法找到对应的后端模块了。

hdrsize 是用户自定义头部的大小，即使把它填成 0，系统还是可以正常找到用户头部的指针的，然后在自己的函数中处理也是可以的。但是内核会报出一个警告提示 Netlink 多了一些字节。所以先定义好你的头部，在这里填上你定义的头部的大小。另外，由于属性列表是放在用户头部的后面，所以如果想使用属性列表，这个值就必须被准确地填充，否则 Netlink 模块找不到对应的属性也就无法解析了。

name 就是给人读的了，对程序没有影响。

genlmsghdr 这个头部需要填充一个 version，这个 version 对应 struct genl\_family 的 version 域。你可以发现，可以同时存在同一个 family 的不同个 version，从而也可以用这种手法实现 family 的复用和提供不同维度的功能。

maxattr 就是该模块支持的最大属性数。每一个属性都是 nla\_policy。family 可以定义一堆属性，请求的时候请求者应该把这些属性放在用户头部的后面。Netlink 会自动解析传递到 do\_it 调用。每一个 family 在定义后都应该调用 genl\_register\_family 向 genetlink 模块注册自己，这样才可以被找到。

我们可以看到在定义 family 的时候没有指定 policy（属性），也就是说属性不是

与 family 相关的，而是与 ops 相关的。

#### 4. ops

ops 一般用来确定模块内的不同操作（当然可以变通使用，概念上就是这么设计的）。有了 family 了还没有定义这个类别下的操作。每一个 family 可以定义多个操作，一个操作并不是一个函数，而是一个函数指针的结构体。因为 Netlink 在设计的时候就把操作分成了两类：set 和 get。或者说是 3 类，还有一个 dump，甚至还可以加上 destroy。所以定义一个 family 的操作就得实现多个函数。

genetlink 为我们封装了细节，但是一个操作还是要实现两个函数（也可以不实现，不实现就不支持对应的 Netlink 的请求 flag，例如 dump），代码如下。

```
static struct genl_opsyy_gnl_ops_tcp_getinfo = {
    .cmd = _C_GETINFO,
    .flags = 0,
    .policy = genl_policy,
    .doit = _doit_getinfo,
    .dumpit = dump_getinfo,
};
```

这个结构体就是一个 ops 操作的结构体。一个 family 可以定义多个这样的结构体，每个结构体都需要调用 genl\_register\_ops 将这个 ops 注册到对应的 family（所以如果 family 没有注册，这个操作就不可能被执行）。

cmd 是一个整数，任意定义，只要保证在一个 family 中没有数与这个数冲突就好。这相当于 ID，就是唯一确定一个 family 中的某个 ops。

policy 就是属性集，这个操作对应的属性集，由于属性列表会传输给 doit 调用，而 doit 也是这个 ops 定义的，所以就相当于给每一个 ops 定义一系列的属性参数。

doit 相当于 set 函数，这个函数的参数是用户发来的请求（被解析好了），模块只需要实现这个函数，但是不能返回值。这个函数当用户请求没有 NLM\_F\_DUMP 标志的时候会被触发调用。

当用户的请求 flag 有 NLM\_F\_DUMP 标志时，调用的就是 dumpit 函数，这个函数没有传入参数，但是可以返回内容。也就是说你可以先调用 doit 来设置内部参数，然后调用 dumpit 来返回。或者是直接使用 dumpit 返回一个列表。用户端的一系列程序，例如 ss 命令都是直接返回列表的。

flags 有 4 种：GENL\_ADMIN\_PERM 表示执行这个 ops 需要 CAP\_NET\_ADMIN 的权限；GENL\_CMD\_CAP\_DO 表示本模块实现了 doit 操作；GENL\_CMD\_CAP\_DUMP 表示本模块实现了 dump 操作；GENL\_CMD\_CAP\_HASPOL 表示本模块实现了属性集。这些 flag 不设置程序也能正常工作，但是设置程序符合标准，是一种符合内核 capabilities 权限控制系统的补充实现。

## 5.4.7 inet\_diag 模块

### 1. 概览

inet\_diag 和 tcp\_diag 是两个模块，但是统一使用 inet\_diag 的接口，inet\_diag 又是使用 Netlink 的接口。要使用这两个模块的功能要首先加载这两个模块，大部分的发行版都是默认加载的（ss 命令就是使用这个）。

使用这套接口获得 TCP 信息，涉及两个问题：请求格式和返回格式。请求格式的代码如下。

```
struct
{
    struct nlmsghdr nlh;
    struct inet_diag_req_v2 r;
} req;
```

因为 Netlink 要求一个通用的 Netlink 头部后面跟具体请求类型对应的数据头部。这里的数据头部使用 inet\_diag\_req\_v2 或者 inet\_diag\_req 都可以，是两种版本的实现，inet\_diag\_req\_v2 更友好一些。

inet\_diag 是 diag 系统中的一部分，它的上面有 sock\_diag，下面有 tcp\_diag。所有的 inet\_diag 都被注册到 sock\_diag 内部的静态数据结构中，每一个 inet\_diag 都是一个方法调用的列表，登记了各种需要的操作。主要的操作有三个：destroy、dump 和 get\_info。

### 2. Netlink 请求头部

inet\_diag 模块是 Netlink 后端的一个子系统，它的请求头部代码如下。

```
struct inet_diag_req_v2 {
```



```

__u8    sdiag_family;
__u8    sdiag_protocol;
__u8    idiag_ext;
__u8    pad;
__u32    idiag_states;
struct inet_diag_sockidid;
};

```

这个是请求 `inet_diag` 的请求，`sdiag_family`、`sdiag_protocol` 与正常的 `socket` 一样的设置 `AF_INET`、`IPPROTO_TCP`。`idiag_states` 就是指 TCP 的连接状态（也可能是 UDP，这取决于你填充的 Netlink 的 `nlh.nlmsg_type=TCPDIAG_GETSOCK;`）。我们这里关注 TCP，填写 TCP 的连接状态，内核对 TCP 连接状态的定义有两个，代码如下。

```

enum {
    TCP_ESTABLISHED= 1,
    TCP_SYN_SENT,
    TCP_SYN_RECV,
    TCP_FIN_WAIT1,
    TCP_FIN_WAIT2,
    TCP_TIME_WAIT,
    TCP_CLOSE,
    TCP_CLOSE_WAIT,
    TCP_LAST_ACK,
    TCP_LISTEN,
    TCP_CLOSING,
    TCP_NEW_SYN_RECV,
    TCP_MAX_STATES
};

enum {
    TCPF_ESTABLISHED    = (1 << 1),
    TCPF_SYN_SENT       = (1 << 2),
    TCPF_SYN_RECV       = (1 << 3),
    TCPF_FIN_WAIT1      = (1 << 4),
    TCPF_FIN_WAIT2      = (1 << 5),
    TCPF_TIME_WAIT      = (1 << 6),
    TCPF_CLOSE          = (1 << 7),

```

```

    TCPF_CLOSE_WAIT    = (1 << 8),
    TCPF_LAST_ACK      = (1 << 9),
    TCPF_LISTEN        = (1 << 10),
    TCPF_CLOSING        = (1 << 11),
    TCPF_NEW_SYN_RECV  = (1 << 12),
};

```

所以，你可以很明显地看出应该用第二个定义。第一个定义是给内部使用的；第二个定义是给外部使用的。第二个定义可以轻松实现不同状态的组合设置。

所以，这里的 `idiag_states` 就用第二个定义来组合设置。如果想要全部状态，就可以任性地使用 `0xff` 来搞定设置。还有一个 `idiag_ext` 域，代码如下。

```

enum {
    INET_DIAG_NONE,
    INET_DIAG_MEMINFO,
    INET_DIAG_INFO,
    INET_DIAG_VEGASINFO,
    INET_DIAG_CONG,
    INET_DIAG_TOS,
    INET_DIAG_TCLASS,
    INET_DIAG_SKMEMINFO,
    INET_DIAG_SHUTDOWN,
};

```

这个 `ext` 可以获得更多种类的信息，包括内存（`ss - m` 参数），如果不填（就是填 0）就是 `INET_DIAG_NONE`，表示什么都不要。也可以看出，同一个请求只能请求一种数据。我们比较关注 TCP 连接的信息，所以使用 `INET_DIAG_INFO`。还有一个是唯一标识一个 `socket` 的域，代码如下。

```

struct inet_diag_sockid {
    __be16  iddiag_sport;
    __be16  iddiag_dport;
    __be32  iddiag_src[4];
    __be32  iddiag_dst[4];
    __u32   iddiag_if;
    __u32   iddiag_cookie[2];
#define INET_DIAG_NOCOOKIE (~0U)
};

```

可以看到, 标识一个 socket 用的不是五元组, 而是源 ip: 源端口; 目的 ip: 目的端口。唯一标示内核中的一个 socket 的 cookie, 这个 cookie 值是在内核中计算 sock 结构体的 sk\_cookie 域得出来的, 一般用户端不需要填充这个域, 在两个字节都放个 INET\_DIAG\_NOCOOKIE 就可以了。

而内核的这个实现只会查找 ESTABLISHED 状态和 LISTEN 状态的连接, 所以想要查询其他状态的 TCP 连接信息的读者可以放弃了。最后那个 socket 绑定的设备也是必须的, 因为内核中的查找也要使用这个信息。但是不是所有的请求都需要填充所有的头部呢? 当然不是, 比如如果你想要全部 dump 整个 TCP 连接表, 就可以不填 sockid 域。

你会发现 iddiag\_src 和 iddiag\_dst 都是 4 个字节的, 这并不是要你输入字符串, 而是要兼容 IPv6, 所以这个接口是 IPv6 和 IPv4 通用的。如果是 IPv4 只需要填充第一个单位就可以了。需要注意的是, 这里地址和端口是网络序的, iddiag\_if 一般是 0。如果你不确定, 先全部填 0, 选项上用 NLM\_F\_DUMP 就可以看到现有的都是怎么存储的了。但是要获得单个的 socket 的信息需要使用 NLM\_F\_ATOMIC, 当然 NLM\_F\_REQUEST 都是必须的。

### 3. 操作种类

总体来说, 所有的 sock\_diag 都只提供一种对外接口, 那就是 dump。但是显然只有这么一种是不够的。inet\_diag 就用这个 dump 接口实现了 dump 和对其他操作的封装。这个 dump 对应的 inet\_diag 模块内部的操作是 inet\_diag\_handler\_cmd 函数。想要获得 Netlink 本身的 dump 信息, 必须得设置 NLM\_F\_DUMP 这个 flag, 代码如下。

```
#define NLM_F_DUMP (NLM_F_ROOT|NLM_F_MATCH)
```

但是在实现功能时我们希望获得 TCP 连接的信息, 由于内核保存 TCP 连接信息的方式是使用 tcp\_hashinfo 全局结构体, 所以本质上就是查询的这个哈希表, 而这个哈希表中只有 ESTABLISHED 和 LISTEN 状态, 所以也查不到别的状态。

内核还有一个 get\_info 接口可以获得很多数据, 但是 sock\_diag 没有对外提供, 其实完全可以对外提供的, 这样就可以获得 TCP 最详细的数据。也就是说现在 inet\_diag 和 tcp\_diag 都支持获得 tcp\_info, 只是 sock\_diag 没有对外提供。而 TCP 通过 getsockopt 对外提供了获得 tcp\_info 结构体的能力。



## 5.4.8 RTNETLINK

RTNETLINK 应该是最常见的 Netlink 类型，其类型为 NETLINK\_ROUTE。在 Netlink 的通用 header nlmsgghdr 之后就跟着 rtnetlink 的 header。这个 header 有几种类型：操纵 MAC 地址的 struct ifinfomsg；操纵 ip 地址的 struct ifaddrmsg；操纵路由表和路由规则的 struct rtmsg；操纵邻居表的 struct ndmsg；操纵 tc 模块的 struct tcmsg。

用这种 socket 可以使用这些种类的功能，也就在通用头部后面附带这些不同功能的结构体。而结构体之外的其他数据，例如要添加一个 ip 地址，添加的 ip 地址的存储位置则是在 struct ifaddrmsg 之后的 struct rtattr 结构里。所有这些功能要携带的数据都是如此。内核还专门定义了一系列操作 struct rtattr 结构体的宏。

下面用一个获取网卡详细信息的案例进行阐述，代码如下。

```
struct rtnl_handle
{
    int fd; //socket 句柄
    struct sockaddr_nl local; //本地地址
    struct sockaddr_nl peer; //对端地址
    __u32 seq; //Netlink 消息的序列号
};

int rtnetlink_open(struct rtnl_handle&rth)
{
    //创建 NETLINK_ROUTE 类型的 socket
    rth.fd = socket(AF_NETLINK, SOCK_RAW, NETLINK_ROUTE);
    if (rth.fd < 0)
    {
        perror("cannot open netlink socket");
        return -1;
    }
    rth.local.nl_family = AF_NETLINK;
    rth.local.nl_groups = 0;
    //绑定到本地地址
    if (bind(rth.fd, (struct sockaddr*)&rth.local, sizeof(rth.local)) <
0)
    {
        perror("cannot bind netlink socket");
    }
}
```



```

        return -1;
    }
    int addr_len = sizeof(rth.local);
    //完善本地地址
    if (getsockname(rth.fd, (struct sockaddr*)&rth.local, (socklen_t*)&addr_len) < 0)
    {
        perror("cannot getsockname");
        return -1;
    }
    if (addr_len != sizeof(rth.local))
    {
        fprintf(stderr, "wrong address lenght %d\n", addr_len);
        return -1;
    }
    if (rth.local.nl_family != AF_NETLINK)
    {
        fprintf(stderr, "wrong address family %d\n",
rth.local.nl_family);
        return -1;
    }
    //初始化作为 Netlink 消息的 seq
    rth.seq = time(NULL);
    return 0;
}

```

上面的代码表示我们定义了一个结构体用于保存本次请求的本次事务，并且对这个中间结构体进行初始化。

```

int rtnetlink_getaddr(struct rtnl_handle* rth){
    rtnl_addr_req req;
    memset(&req, 0, sizeof(req));
    req.hdr.nlmsg_len = NLMSG_LENGTH(sizeof(rtnl_addr_req)); //设置
消息长度
    req.hdr.nlmsg_type = RTM_GETADDR; //设置类型为获取地址
    req.hdr.nlmsg_flags = NLM_F_DUMP | NLM_F_REQUEST; //设置 flags 为
请求和 dump
    req.hdr.nlmsg_seq == ++rth->seq; //构造 seq

```





```

    req.addrmsg.ifa_family = AF_INET ;
    req.addrmsg.ifa_prefixlen = 32 ;
    // 这个就是对应的网卡的 id, 这个 id 可以通过 cat /sys/class/net/lo/ifindex 获
    得对应网卡的 id

    req.addrmsg.ifa_index = 1 ;
    req.addrmsg.ifa_scope = 0 ;
    if( send(rth->fd, &req, req.hdr.nlmsg_len, 0)<0){ //发送消息
        perror("send");
        return 1;
    }
    cout<<"receiving message"<<endl;
    char buf[8192];
    int status = recv(rth->fd, buf, sizeof(buf), 0); //接收并判断
    返回消息

    if (status < 0) {
        perror("recv");
        return 1;
    }
    printf("status = %d\n", status);
    if (status < 0) {
        cerr<<"errno = "<< strerror(errno)<<endl;
        return -2;
    }
    if (status == 0) {
        printf("EOF\n");
        return -3;
    }
    //解析并处理得到的网卡的详细信息
    for( struct nlmsgshdr* h = (struct nlmsgshdr *)buf; status > sizeof(struct
    nlmsgshdr);){
        int len = h->nlmsg_len;
        int req_len = len - sizeof(struct nlmsgshdr);
        if (req_len<0 || len>status) {
            printf("error\n");
            return -1;
        }
        if (!NLMSG_OK(h, status)) {

```





```

        printf("NLMSG not OK\n");
        return 1;
    }
    //获取到实际的数据结构体指针和属性指针
    struct ifaddrmsg* rcv_addrmsg = (struct ifaddrmsg
*)NLMSG_DATA(h);
    struct rtattr* rtatp = (struct rtattr *)IFA_RTA( rcv_addrmsg );
    //从网卡的 index 获取到 name
    char iface_name_buf[IF_NAMESIZE];
    if_indextoname( rcv_addrmsg->ifa_index, iface_name_buf);

    printf("Index Of Iface= %d, name:%s\n",rcv_addrmsg->ifa_index,
iface_name_buf);
    //解析属性数据
    int rtattrrlen = IFA_PAYLOAD(h);
    for (; RTA_OK(rtatp, rtattrrlen); rtatp = RTA_NEXT(rtatp, rtattrrlen)) {
        if(rtatp->rta_type == IFA_ADDRESS){ //网卡 ip 属性
            struct in_addr *in = (struct in_addr
*)RTA_DATA(rtatp);

            printf("addr0: %s\n", inet_ntoa(*in) );
        }
        if(rtatp->rta_type == IFA_LOCAL){ //local ip 属性
            struct in_addr *in = (struct in_addr
*)RTA_DATA(rtatp);

            printf("addr1: %s\n",inet_ntoa(*in));
        }
        if(rtatp->rta_type == IFA_BROADCAST){ //广播地址属性
            struct in_addr *in = (struct in_addr
*)RTA_DATA(rtatp);

            printf("bcataddr: %s\n",inet_ntoa(*in));
        }
        if(rtatp->rta_type == IFA_ANYCAST){ //任播地址属性
            struct in_addr *in = (struct in_addr
*)RTA_DATA(rtatp);

            printf("anycastaddr: %s\n", inet_ntoa(*in));
        }
    }
}

```



```
//更新以处理下一个消息
status -= NLMSG_ALIGN(len);
h = (struct nlmsg_hdr*)((char*)h + NLMSG_ALIGN(len));
}
```

通过这两个函数的连续调用，就可以获取到网卡的全部信息了。RTNETLINK 还有很多丰富的功能和选项，所能获得的信息远不止如此。



## 6

## 第 6 章

## Linux 内核内存管理

## 6.1 内存模型

## 6.1.1 内存模型概览

很多小型操作系统，例如 eCos、VxWorks 等嵌入式系统，系统中的程序所采用的地址就是实际的物理地址。这里所说的物理地址是 CPU 所能见到的地址，至于这个地址如何映射到 CPU 的物理空间，映射到哪里，都取决于 CPU 的种类（例如 MIPS 或 ARM），一般是由硬件完成的。对于软件而言，启动时 CPU 就能看到一片物理地址（线性地址）。但是一般比嵌入式系统大一点的系统，刚启动时看到的已经映射到 CPU 空间的地址并不是全部的可用地址，需要用软件去想办法映射可用的物理存储资源到 CPU 地址空间。很多硬件中都带有 Security Boot 的启动固件，这个固件也可以被用来完成内存映射。

通常 CPU 可见的地址是有限制的，32 位的 CPU 最多可见 4GB 的物理空间，64 位的 CPU 可见的物理空间会更大。所以目前应用的 64 位系统可能不需要考虑物理内存 CPU 可见物理空间的问题，然而 32 位的系统基本都是要考虑的，这就诞生了一个需求——动态映射。





在 Linux 系统中，例如 x86 架构，由于 CPU 可见的 3GB 的空间给了用户程序，内核仅留下了 1GB 空间，而存储的映射都要映射到这 1GB 内存中，所以大于 1GB 的内存在内核中不使用动态映射都无法访问。简单地说，就是当需要一个空白内存页的时候，动态地将某个物理内存映射到一个地址，需要换下已经映射过的物理地址，重新映射新的物理地址到这个线性地址。

另外一个需求是，现在的系统通常不止跑一两个程序，而每个程序又都可以看见和操作完整的地址，这样安装别人发布的进程就是一个危险性很高的操作。嵌入式系统的内存资源控制相对容易处理，但 PC 机难以处理这个问题。因此，每个程序在程序可见的地址空间隔离是非常有必要的，于是有了虚拟的程序地址空间。每个进程见到的地址范围都是一样的，然而访问同一个地址返回的数据却不一定是一样的。

无论是用户程序还是内核程序，都需要使用内存，所以如何高效地分配和回收内存就是一个很重要的话题。在实际的需求中，用户可以申请内存，但申请的内存不一定会使用，因此内核也可以为用户预留内存，只是在其真正使用的时候才分配。这种内核机制叫作 `over_commit`，就是内核可以为应用程序分配大于实际拥有的内存量。

Linux 内核会使用大量的空间缓存磁盘中的文件，这部分内存会占用几乎所有的可用内存。当用户程序对内存有需求的时候，Linux 就会回收一部分内存，用来满足用户的需求。所以表面看 Linux 系统的可用内存几乎永远为 0，然而申请内存又通常可以成功。

这些内存针对各个功能的需求而设计的机制共同组成了 Linux 的内存管理机制，离开具体功能的内存管理机制是没有意义的。因此内存管理主要有三个需求：动态的物理内存的管理、隔离的用户地址空间管理以及内存的分配和回收。

## 6.1.2 内存组织方式

### 1. x86组织

Linux 内核内存以页为单位，但整体被组织为 zone（区域），一共有 3 个 zone：DMA、Normal 和 High。有的硬件架构的 DMA 系统只能访问一部分地址（如 Intel 的 DMA 只能访问低于 16M 的地址），但是例如在 MIPS 平台上，在编译内核的时候就可以关闭 DMA zone，关闭一个 zone 可以有效加速内存回收算法的执行速度。有



的系统可用的物理内存远远超过了 CPU 可见的内存空间，如 32 位的 CPU 对于 4GB 以上的内存就无法全部静态映射。但是由于 Linux 的虚拟内存机制，内核能使用的所有空间仅有 1GB（在一些架构中可变）。

一般来说，Linux 内核按照 3:1 的比例来划分虚拟内存（x86）：3 GB 的虚拟内存用于用户空间，1GB 的内存用于内核空间。当然，有些体系结构，如 MIPS 使用 2:2 的比率来划分虚拟内存：2 GB 的虚拟内存用于用户空间，2 GB 的内存用于内核空间。另外像 ARM 架构的虚拟空间是可配置的（比如 1:3、2:2、3:1）。

以 x86 为例，Linux 中内核使用 3~4GB 的线性地址空间，也就是说内核总共只有 1GB 的地址空间可以用来映射物理地址空间。但是，如果在内存大于 1GB 的情况下内核范围内的线性地址就不够用了。为此内核引入了一个高端内存的概念，把 1GB 的线性地址空间划分为两部分：小于 896MB 的物理地址空间称之为低端内存，这部分内存的物理地址和用户端可见的 3GB 大小的内存空间的开始位置的线性地址是一一对应映射的，也就是说内核使用的线性地址空间（VA）3GB（3GB+896MB）和物理地址空间（PA）0~896MB 一一对应，也就是 `PAGE_OFFSET=0xC0000000`；剩下的 128MB 的线性空间用来映射剩下的大于 896MB 的物理地址空间，即我们常说的高端内存区，这部分空间需要 MMU 通过 TLB 表建立动态的映射关系。

在 Linux 下 x86 的 32 位系统，真正可以静态映射的内存只有 896MB。当内存大于 1GB 时就需要使用高端内存了，否则大于 1GB 的内存就无法使用。所以三个内存的 zone，前 16MB 的 DMA 区域对应内核空间的 0~16MB；Normal 区对应 16~896MB；High 区对应 896MB~1GB 的动态区，可用大小实际是可变的。从这里可以看出，如果不需要 DMA 区（DMA 无限制），则该区可以删除。如果内存不超过 896MB，highmem 区也可以删除（但是很多嵌入式系统即使没有超过 896MB，也需要使用 highmem 的动态映射能力）。

因为内核在响应请求分配空间时是在 3 个区中都分配的，优先是 Normal 区，回收的时候也是 3 个区都执行回收的。如果能去掉一个区，对于很多内存操作来说就能减少很多执行所付出的代价。

## 2. MIPS 组织

在 MIPS 32 CPU 中不经过 MMU 转换的内存窗口只有 `kseg0` 和 `kseg1` 的 512MB 的大小，而且这两个内存窗口映射到同一 0~512MB 的物理地址空间。其余的 3GB 虚拟地址空间需要经过 MMU 转换成物理地址，这个转换规则是由 CPU 厂商实现



的。换句话说，在 MIPS32 CPU 下面访问高于 512MB 的物理地址空间，必须通过 MMU 地址转换，即按  $VA=PA+PAGE\_OFFSET$  公式映射的空间最大只有 512MB，其中  $PAGE\_OFFSET=0x80000000$ ，而在 Linux 中 MIPS 32 只使用其中的 256MB。

MIPS 在 highmem 使用过程中需要注意两个问题，一是要考虑由 highmem 带来的整个系统性能和稳定性之间的平衡；二是 highmem 不支持 cache aliases，而 MIPS 架构必须要求指令对齐，很多问题都由此而起。

## 6.2 申请和释放内存

我们知道现在的操作系统的内存都是按页划分组织的，不同的系统会在页之上添加页组等概念。但是内核内存管理的基本单元是页，所以最基本的内容就是内核如何管理页。

### 6.2.1 高端内存

高端内存映射有 3 种方式：临时映射空间、长久映射空间、非连续映射地址空间。

#### 1. 临时映射空间

固定映射空间是内核线性空间中的一组保留虚拟页面空间，位于内核线性地址的末尾，即最高地址部分。其地址编译时确定用于特定用途（如 VSYSCALL 系统调用，MIPS 的 cache 着色）。由枚举类型 `fixed_addresses` 决定，固定映射空间位于 `FIXADDR_START` 到 `FIXADDR_TOP` 之间，有一部分用于高端内存的临时映射。这块空间具有以下特点：每个 CPU 占用一块空间；可以用在中断处理函数和可延迟函数的内部，从不阻塞，禁止内核抢占；在每个 CPU 占用的那块空间中，又分为多个小空间，每个小空间的大小是 1 个 page，每个小空间用于一个目的，这些目的定义在 `kmap_types.h` 中的 `km_type` 中。

当要进行一次临时映射时，需要指定映射的目的。根据映射目的可以找到对应的小空间，然后把这个空间的地址作为映射地址。这意味着一次临时映射会导致以前的映射被覆盖。代码如下。

```
void *kaddr = kmap_atomic(page);
```





```
memcpy(kaddr, data, len);  
kunmap_atomic(kaddr);
```

接口函数是 `kmap_atomic`、`kunmap_atomic`。使用从 `FIX_KMAP_BEGIN` 到 `FIX_KMAP_END` 之间的物理页。

## 2. 长久映射空间

长久映射空间是预留的线性地址空间，是访问高内存的一种手段。使用方式是先通过 `alloc_page()` 获得高端内存对应的 `page`，然后内核从专门为此留出的线性空间分配一个虚拟地址，在 `PKMAP_BASE` 到 `FIXADDR_START` 之间。

接口函数是 `void*kmap(struct*page)`、`void kunmap(struct*page)`。该接口函数在高、低内存都能使用，可以睡眠，数量有限。对于不使用的 `page`，应该及时从这个空间释放（解除映射关系）。

随着内核的发展，人们发现长久映射的需求都可以由临时映射来解决，所以目前 `kmap_atomic` 也在逐渐替代 `kmap` 调用。

## 3. 非连续映射地址空间

非连续映射地址空间适用于不频繁申请释放内存的情况，这样不会频繁地修改内核页表。总的来说，内核主要在以下情况使用非连续映射地址空间：映射设备的 I/O 空间；为内核模块分配空间；为交换分区分配空间。

每个非连续内存区都对应一个类型为 `vm_struct` 的描述符，通过 `next` 字段，这些描述符被插入到一个 `vmlist` 链表中。在这种方式下高端内存使用起来简单，因为通过 `vmalloc()` 就可能从高端内存获得页面。接口函数是 `vmalloc(vfree)`，物理内存（调用 `alloc_page`）和线性地址同时申请，物理内存是 `__GFP_HIGHMEM` 类型（分配顺序是 High 区、Normal 区、DMA 区），可见 `vmalloc` 不仅可以映射 `HIGHMEM` 页框，而且它的主要目的是为了将零散的、不连续的页框拼凑成连续的内核逻辑地址空间。

## 6.2.2 设备内存映射

`ioremap`（`iounmap`）可以用于分配 I/O 映射空间，将 I/O 的寄存器和 `ram` 空间映射到内核空间，使 CPU 可以直接访问对应的线性地址。但是很多驱动都不会使用这



个接口，而是自己做线性地址到物理地址的转换，只需要几行代码即可。这个函数真正的价值在于在调用了该函数完成映射之后还可以调用 `remap_page_range`，将设备内存 `ram` 或 `rom` 直接映射到用户空间中，这样用户端就可以直接操作设备寄存器了。

启动时 `ioremap` 调用还没有就绪，这时内核就提供一个早期的 `ioremap` 调用，叫作 `early_ioremap`。

### 6.2.3 启动时内存的申请和释放：bootmem

Linux 启动时的各个模块也有申请和释放内存的需求，但是此时内核的内存模型还没有建立好。于是 Linux 就提供了一个专门用在此时的内存接口 `bootmem`，这个接口很简单，以页为单位，简单地搜索满足需求的连续页空间分配，并且可以应对物理上不连续的存储体。

这个内存机制还有一个最广泛的使用技巧，就是分配超大额连续内存。因为在系统启动之前，这个需求是容易满足的，但是在系统启动之后，由于模块众多，内存使用频繁换页，连续的物理内存很难得到，在启动时直接通过 `bootmem` 接口预留连续的物理内存，留给后续使用是不二的选择。在内核完全启动后，`bootmem` 机制不再有效。

### 6.2.4 Mempool

Linux 通过 `slab` 伙伴系统分配内存，这种方式虽然可以做到在内存不足时进行回收来获得内存，但是没办法保证关键路径内存获得的稳定性。在用户端编程中也经常用到一个手法，就是内存池，在程序运行开始时就分配一些独占内存。在程序运行过程中，可以确保在一定的时间内获得指定的内存。这是以牺牲使用效率换取稳定服务的做法。接口代码如下。

```
mempool_t *mempool_create(intmin_nr, mempool_alloc_t*alloc_fn,
    mempool_free_t *free_fn,void *pool_data); //内存池的创建
void *mempool_alloc(mempool_t *pool, intgfp_mask); //从内存池中分配对象
void mempool_free(void *element, mempool_t *pool); //将对象放回内存池中
intmempool_resize(mempool_t *pool, intnew_min_nr, intgfp_mask); //重新
调整内存池大小
```



## 6.2.5 CMA（连续内存分配器）

CMA（连续内存分配器），在有这个之前，想要预留一大块连续的内存，基本只能在启动的时候使用 `bootmem` 预留，这样预留带来的代价就是 Linux 系统启动后，这部分内存对于内核不可用。而用户预留的内存又不一定一直在使用，从而导致内存的利用率偏低。

## 6.2.6 伙伴算法

内存存在底层是以页为单位分配的，上层一些的分配器，如内核的 `slab`、用户控件的 `malloc` 等都是在后台先申请了足够的页之后再对用户进行分配。这样后台关于如何申请页就有很多种思路，这些思路主要围绕两个标准展开：如何最快、如何碎片最少。

伙伴算法被广泛使用，该算法的核心思想是把内存提前分为大小不同的一系列内存块，当申请内存的时候返回最贴近需求内存大小的内存块，没有合适大小的时候就可能拆分成更大的内存块。通过提前安排，在牺牲内存利用率的前提下，尽可能地实现非碎片化。这个思想也不是一直有效，后来人们还加入了内存页的回收类型属性：可回收、可移动、不可回收。相当于定期对磁盘进行磁盘整理来让不连续的空闲内存块重新连续起来。由于用户程序使用的内存页都是通过动态映射来的，所以后台只需要替换一下映射就能实现对用户程序透明的页面置换，所以这种做法也是不错的。

除了在分配上注意不产生碎片外，内核也会定期回收已经分发出去的页面。合理分发加上有效回收构成了 Linux 内存管理的核心。

## 6.2.7 slab

内核中有很多常用的结构体，如果使用简单的结构体并根据大小进行动态分配，将会频繁地搜索链表，显然使用 `pool` 思想更合适。但由于常用的结构体有很多，不可能为每一个结构体定义一个池类型，合理的做法应该是尽可能地通用，这个被设





计出来的结构体池就是 slab——内存管理机制。

slab 内存管理机制的得名是由于其将一种结构体的内存池命名为 slab，内核中同时存在多个 slab，分别是不同的常用结构体的池。为了适应 SMP，让每一个 CPU 都管理一系列独立的 slab。

但是 slab 在 NUMA 上的适应能力不行，slub 在 slab 的基础上增加了 NUMA 的适应能力，还精简了 slab 的结构体，提高了 slab 的效率，但与 slab 提供的调用接口是一样的。

而 slob 则是精简版的 slab，增加了内存分配的碎片化概率，本质上是降低了效率，但是需要更少的资源开销（内存和 CPU），所以大部分 slob 是应用在嵌入式系统中的，但是目前的嵌入式系统的计算能力也普遍强大，所以 slob 基本退出历史舞台。

## 6.2.8 用户端内存管理基础组件

在 glibc 中使用的 malloc 并不是唯一可用的内存管理方法，此外还有 Bionic 的 dlmalloc，Google 的 tcmalloc，以及被普遍认为最强的 jemalloc。jemalloc 的核心思想是将内存池分为 3 级，每个线程都有自己的内存池，向上有一些 large 的内存池，最上面是 huge 内存池。而 tcmalloc 管理的是一系列内存池，每个线程都会发展出与某个内存池的亲和度。所以 jemalloc 适合线程数比较固定的场合，而 tcmalloc 适合线程数变动比较大的场合。

一般我们在后台开发经常用到的是 jemalloc，例如笔者曾经使用老版本的 golang 做云存储系统，在底层内存分配特别频繁的时候，glibc 明显有无法处理的情况，或者是内存泄漏、效率显著降低的情况。此时换用 jemalloc 就能解决问题。现代协程语言，例如 Golang，线程变化都不会太频繁，而 jemalloc 在多核的情况下表现非常优秀，所以大部分情况下 jemalloc 都会是理想的选择。

glibc 的内存分配出现泄漏是每一个程序员的痛。如何发现这种内存泄漏是一个很重要的话题。glibc 本身提供了一个 mtrace 命令用来查看 malloc 内存的申请和释放，使用 mtrace 命令就能看到内存什么时候申请、什么时候释放。只需要在进程中添加一行代码（setenv("MALLOC\_TRACE","output",1)）就会产生一个内存文件，mtrace 就能读取这个文件从而输出报告。而大部分的内存泄漏检测都会有更加成熟的工具，使用量最大的应该是 valgrind 套件。



## 6.3 内存组件

### 6.3.1 内存回收算法 (PFRA)

由于 Linux 内核的思路是内存都要尽量使用，所以内存回收是一个非常频繁的操作。并且只要用户进程不释放内存，内核除非 OOM (Out Of Memory)，否则就无法从应用的内存中回收资源。我们经常看到在形容内存页的时候有命名页和匿名页两种，命名页对应应在内存中缓存的文件，这部分是重点回收的，当然还有其他可以回收的地方。

在用完所有空闲内存之前，就要执行页框回收算法。这很正常，因为我们执行算法必须要使用内存，如果等待内存全部用完了再启动算法就会卡死。内核中有 3 个水线值来控制什么时候启动算法，回收到什么比例结束，到了什么最低值就要阻塞进程进行强制回收和 OOM，这 3 个水线值都在 `/proc/sys/vm/min_free_kbytes` 中设置一个值，内核会根据这个值实际计算出这 3 个值，但是计算的算法比较一般，用户可以自己修改内核代码来改变这 3 个值。

`kswapd(mm/page_alloc.c)` 根据 `min_free_kbytes` 计算每个 zone 的 3 个 watermark (min、low、high)，当系统可用内存低于 `watermark[low]` 的时候，就会叫醒 `kswapd`。如果 `kswapd` 回收的内存不如上层申请内存的速度快，使可用内存降至 `watermark[min]` 时，就会触发直接内存回收机制。而这种方式会阻塞应用程序。

所以，例如 `samba` 在使用时有可能出现堵塞问题的原因可能就是 `kswapd` 回收内存的速度慢于使用内存的速度，从而触发了直接内存回收机制，导致 `samba` 阻塞。

页框算法保存一定的空闲页框，并使内核可以安全地从缺少内存的情形中恢复过来。这个页框回收算法的关键参数都可以在 `proc` 文件系统中进行调整。在很多情况下，这会成为影响内核性能的关键。我也曾经通过去掉 DMA zone、算法参数优化和修改一些细微的算法就使得一个嵌入式板的 USB 传输速度从 4Mb/s 提高到了 30Mb/s。

PFRA 的目标就是获得页框并使之空闲。PFRA 按照页框所含内容，以不同的方式处理页框。页框被分为不可回收页、可交换页、可同步页和可丢弃页。其中 PFRA 可以回收除不可回收页之外的其他页。

内核检查页面回收分为周期性检查和内存不足时的阻塞检查。周期性的检查是



由后台运行的守护进程 `kswapd` 完成的。该进程定期检查当前系统的内存使用情况，当发现系统内空闲的物理页面数目少于特定的阈值时，该进程就会发起页面回收的操作。

直接页面回收会阻塞当前进程的执行，直到回收了足够的内存再唤醒。因此在例如 `samba` 拷贝众多小文件的时候，如果忽然卡住，很有可能是这个原因（当然还有可能是 `TCP` 的原因）。

如果操作系统在进行了内存回收操作之后仍然无法回收到足够多的页面以满足上述内存要求，那么操作系统只有最后一个选择，那就是使用 `OOM (out of memory)` killer，它从系统中挑选一个最合适的进程并“杀死”它，同时释放该进程所占用的所有页面。`OOM` 子系统的打分方法也可以在 `proc` 中调整。早期的内核版本是允许在 `proc` 文件系统中关闭 `OOM` 功能的，现在的内核都不允许了。在内核编译的时候还可以配置 `OOM` 时的表现状态，可以配置为 `panic`，在 `OOM` 发生的时候 `kernel` 直接停机。这在很多系统中是比较常见的做法。

上面介绍的内存回收机制主要依赖 3 个字段：`pages_min`、`pages_low` 及 `pages_high`。每个内存区域在其区域描述符中定义了这样 3 个字段，这 3 个字段的具体含义如下。

- `pages_min`: 区域的预留页面数目，如果空闲物理页面的数目低于 `pages_min`，那么系统的压力会比较大。此时，内存区域中急需空闲的物理页面，页面回收的需求非常紧迫。
- `pages_low`: 控制进行页面回收的最小阈值，如果空闲物理页面的数目低于 `pages_low`，那么操作系统的内核会开始进行页面回收。
- `pages_high`: 控制进行页面回收的最大阈值，如果空闲物理页面的数目高于 `pages_high`，则内存区域的状态是理想的。

### 6.3.2 其他内存功能组件

- `Frontswap`: 可以为 `swap` 提供一个更快的缓存层，通常位于内存中。
- `Kasan`: 发现内核内存错误的功能。
- `hwpoison_inject`: 允许内核标记一个页为有毒的，被标记页的创造进程会被杀掉，进行消毒。



- **Kmemcheck**: 内核代码访问非法的内存地址（如访问未初始化的内存，访问已经释放的内存）的检查。
- **Kmemleak**: 内核泄漏检测方法，其类似于跟踪内存收集器。当独立的对象占用的内存没有被释放时，就记录在 `/sys/kernel/debug/kmemleak` 中。
- **Ksm**: 内存中有很多页的内容是一样的，这些页没有必要在内存中有多份拷贝。这个机制就是让这些页用一个写保护的页取代。这个机制只合并匿名页，不合并文件缓存页（匿名页是指所有没有缓存文件的页）。
- **Memory hotplug**: 支持内存热插拔。
- **NUMA**: 支持跨 CPU 离散内存。
- **balloon\_compaction、compaction**: 把稀疏的页数据整理紧凑，这样可以占用更少的页。

### 1. mmap 和 mlock

**mmap** 是一个比较常见的研究用途的系统调用，普通用户编程几乎没有使用它的。这个系统调用在处理文件时有很多不方便之处，只有对业务模型是多个进程，同时访问一个文件，出于效率考虑，愿意自己去完成一系列的内存保护和刷新的高级服务器端的用户才会使用。例如当映射的文件大小会变化，而 **mmap** 映射是固定大小时就会带来困扰。**MAP\_POPULATE** 选项可以配合映射文件启动内核的预读机制，以加速后续文件的读取。

**mmap** 不但可以 **map** 文件到内存，还可以用来直接申请一大块内存，使用 **MAP\_ANONYMOUS** 作为 **flag** 就可以得到一大块初始化为 0 的内存。当然，也可以加上 **MAP\_UNINITIALIZED** **flag** 使内存不必清 0。一般的上层用户也不会想用这个功能，一般是 C 语言底层程序员，使用这个功能的时候就会感觉到它的强大。因为用这种方法获得的内存直接绕过了 **malloc**，并且可以与其他进程直接共享。重要的是，当不再需要这块内存时，调用 **unmap** 可以直接把内存还给内核，其他进程可以立即使用该内存。而 **malloc** 则会还到自己的缓存里，还是当前进程独占。这对于内存敏感的 C 程序是一个重要特性。如果再配合使用 **MAP\_HUGETLB** 就可以在大页内存上直接申请内存。

如果使用 **MAP\_LOCKED** 就有 **mlock** 的效果，可以把内存限制在 **ram** 内，而禁止被交换到 **swap** 里，这对于性能敏感的程序很有意义。

```

struct bdi_writeback_congested *wb_congested;
wait_queue_head_t wb_waitq;

struct device *dev;
struct timer_list laptop_mode_wb_timer;
};

```

从以上的数据结构很容易看出 BDI 的全部功能。没有 BDI 的磁盘设备也是可以正常工作的。但是 BDI 为所有的磁盘设备提供了高层次的数据缓存功能。这个缓存层位于文件系统的下层和通用块层的上层，显然这个缓存功能属于内存管理的一部分。

相对于内存来说，BDI 后端设备（比如最常见的硬盘存储设备）的读写速度是非常慢的，因此为了提高系统的整体性能，Linux 系统对 BDI 设备的读写内容进行了缓冲，那些读写的数据会临时保存在内存里，以避免每次都直接操作 BDI 设备，但这就需要在一定的时机（比如每隔 5 秒、垃圾数据达到一定的比率时等）把它们同步到 BDI 设备中，否则长久地呆在内存里容易丢失（比如机器突然宕机、重启），而进行间隔性同步工作的进程之前名叫 `pdflush`，但后来 Kernel 2.6.2x/3x 对此进行了优化改进，产生有多个内核进程，`bdi-default`、`flush-x:y` 等。

关于以前的 `pdflush` 不再多说，我们这里只讨论 `bdi-default` 和 `flush-x:y`，这两个进程（事实上 `flush-x:y` 为多个）的关系为父与子的关系，即 `bdi-default` 根据当前的状态 Create 或 Destroy `flush-x:y`，`x` 为块设备类型；`y` 为此类设备的序号。如有两个 TF 卡，则分别为 `flush-179:0` 和 `flush-179:1`。

`flush` 内核线程为了回写磁盘在 BDI 数据结构中定义的一个 `writeback` 对象的磁盘的页存储，该对象是对 `writeback` 内核线程的描述，并且封装了需要处理的 `inode` 队列。在 BDI 数据结构中有一条 `work_list`，该 `work` 队列维护了 `writeback` 内核线程需要处理的任务。如果该队列上没有 `work` 可以处理，那么 `writeback` 内核线程将会睡眠等待。代码如下。

```

struct bdi_writeback {
    struct backing_dev_info *bdi;    /* 属于 BDI 指针 */
    unsigned long state;
    unsigned long last_old_flush;
    struct list_head b_dirty;
    struct list_head b_io;
    struct list_head b_more_io;
};

```

```

structlist_headb_dirty_time;
spinlock_tlist_lock;
structpercpu_counter stat[NR_WB_STAT_ITEMS];
structbdi_writeback_congested *congested;
unsigned long bw_time_stamp;
unsigned long dirtied_stamp;
unsigned long written_stamp;
unsigned long write_bandwidth;
unsigned long avg_write_bandwidth;
unsigned long dirty_ratelimit;
unsigned long balanced_dirty_ratelimit;
structfprop_local_percpu completions;
intdirty_exceeded;
spinlock_twork_lock;
structlist_headwork_list;
structdelayed_workdwork;
structlist_headbdi_node;
};

```

以上的数据结构中大部分从拼写上就可以大概猜出其含义，但是详细介绍每个域对于整体的理解并没有任何意义。writeback 对象封装了任务 dwork 以及需要处理的 inode 队列 bdi\_node。当需要刷新 inode 时，可以将该 inode 挂载到 writeback 对象的 b\_dirty 队列上，然后唤醒 writeback 线程。在处理过程中，inode 会被移到 b\_io 队列上进行处理。

### 1. cleancache 与 zcache

位于 vfs 层的一个接口定义，用于以页的形式缓存磁盘数据。我们知道 BDI 已经完成了相同的工作，为何还需要 cleancache 再来完成一遍呢？cleancache 是后来引进的，引进它的目的是为了虚拟化，它可以在多个虚拟机中缓存同一份数据，达到节省空间的目的。所以，在单机的情况下使用此机制无意义。然而 cleancache 有另一个后端，就是存储缓存页使用压缩的方式，叫作 zcache，在这种情况下就比 BDI 有优势了。

### 2. fadvise

我们知道读取文件系统的文件是有缓存存在的。但是用户可以对如何使用这个



缓存提出建议。这个提意见的需求是因为使用文件的人是用户程序，用户才知道他自己将如何使用一个文件，而例如要对文件进行顺序读取，内核对文件的缓存可能就是全面的，但如果是顺序读取，内核对文件的缓存就是顺序的（读了后面丢了前面）。通过 `fcntl` 接口，用户就可以告诉内核这个信息，使缓存系统更加高效地工作。

但理论如此，目前的实现却比较简单，例如顺序读取，目前仅仅是把该文件预读的页扩大一倍。还有一个最重要的原则，用户永远只能建议内核，内核是否照做取决于内核。

# 7

## 第 7 章

### 安全

#### 7.1 概览

Linux 从 UNIX 和 POSIX 那里继承了最基本的安全机制：用户、文件权限、进程 capabilities。但是仅有这些是不够的，很多第三方实现了新的机制，并通过补丁的形式提供，如安全增强 Linux (SELinux)、域和类型增强 (DTE)，以及 Linux 入侵检测系统 (LIDS) 等，但是没有哪个完全胜出，所以都没有进入主内核代码。

Linux 内核的创始人 Linus Torvalds 表示 Linux 内核确实需要一个通用的安全访问控制框架，但他指出最好是通过可加载内核模块的方法实现，这样可以支持现存的各种不同的安全访问控制系统。因此，Linux 安全模块 (LSM) 应运而生。SELinux、DTE、LIDS、AppArmor、SELinuxSmack、TOMOYO Linux、Openwall 等都是通过 LSM 框架提供了自己的服务。

LSM 框架的最大优点是对内核的改变少，类似 netfilter 的 hook 机制，将安全点做成 hook，各种安全机制对安全的限制策略都通过 hook 点去实现，这样就兼容大部分的安全方案。如果你发现某个安全模式不好用，可以换其他的内核模块来提供内核安全防护。

我们知道安全防护有保护和审计两个维度，LSM 模块大部分的作用是保护，控制是否允许访问，而审计则是交给了具体的安全模块去完成。

发中常说的加盐的思路是一样的。

CRC 则是非常常用的摘要算法，碰撞概率太高，但是计算速度快，所以一般用在对简单数据的摘要计算上，例如网络通信中的数据包。

摘要算法在内核中几乎都支持，但是很多文件系统和比较大的模块会自己实现一套摘要算法。内核中 `crypto/hmac.c` 文件中就以模块的形式提供了 `hmac` 的支持。如果你仔细看 `crypto` 这个内核目录，会发现更多的其他算法的实现支持。

### 7.2.3 加密

有加密就有解密。业界一般根据加密和解密的密钥是否一样，将加密算法分为对称性的和非对称性的。对称性的要另外找私密的渠道进行密钥分发，所以在使用上很不方便。由于非对称性加密和解密密钥不一样，所以可以公布一个加密的密钥作为公钥，任何人都可以使用这个密钥对数据进行加密，但是只有私钥的持有者可以解密。而反过来只有自己能加密、别人都只能解密的方式就可以用来验证这个数据确实是由自己发出的。也就是说公布了加密密钥可以用来加密通信，公布了解密密钥就可以用来认证。

常见的对称性加密算法有 DES、3DES、AES。DES (Data Encryption Standard) 是早期的数据加密标准，它速度较快，适用于加密大量数据的场合，可以用穷举法破解（但得有非常强的计算能力）；3DES (Triple DES) 是基于 DES 的，对一些数据用 3 个不同的密钥进行 3 次加密，强度比 DES 更高；AES (Advanced Encryption Standard) 是下一代的加密算法标准，它速度快，安全级别高。在实际实现上 DES 类是分组的，AES 是流式的。由于各个分组不发生交互运算，所以在安全强度上比较脆弱，流式之前的运算结果会影响下面的计算，所以目前普遍认为流式比分组式更安全（但其实就是要穷举多久的问题）。

常见的非对称性加密算法有 RSA、Elgamal、背包算法、Rabin、D-H、ECC (椭圆曲线加密算法)。目前使用最广泛的非对称性加密算法是 RSA。RSA 算法基于一个十分简单的数论事实：将两个大素数相乘十分容易，但是想要对其乘积进行因式分解却极其困难，因此可以将乘积公开作为加密密钥，而只有自己知道，其他人很难算出来的因式分解就是解密密钥了。在这个算法中，公开的只能是加密密钥。这也就决定了这个算法不能简单地用于认证，但通过一系列双向交互就可以实现认证



和数字签名。其他算法也都是类似的数学问题的求解，如椭圆双曲线。

对称密钥的计算速度明显快于非对称密钥，缺点是缺少安全的密钥分发渠道。所以业界普遍使用非对称加密来做认证和生成对称密钥（也叫会话密钥），然后双方使用对称密钥进行通信。而由于对称密钥可以靠穷举法进行破解，所以一般的会话管理都会自动更换会话密钥，如此看来一个定期更换会话密钥的通信是安全的，但是安全永远是相对的。

## 7.2.4 认证

前面说过，加密和认证在数学上用的是同样的算法，只是用法不同，认证主要使用的是非对称性加密。

由于采用素数积的方式只能公布加密密钥，所以通过公布解密密钥来做到认证的做法不可取。但是通过加入第三者就可以，这就是密钥管理系统，最常见的是 Kerberos 系统。认证过程一般有一个认证中心，被这个中心认证过的证书（X.509）就是可以信任的。这个认证过程本质上就是一些加密、解密的运算。以下是一个示例。

首先是任何一个想要申请认证的企业都需要一个服务器（我们假定是 Linux），需要首先在其服务器上生成自己的私钥，并且加密，代码如下。

```
openssl genrsa -out private.key 2048 //生成自己的私钥
openssl rsa -in private.key -des3 -out encrypted.key //对私钥进行加密
```

这样就可以生成加密过的私钥，可以利用这里的密钥做到 SSH 登录的无用户但有密码登录。也就是 SSH 配置为的只是识别密钥登录，但是这个密钥却是加密过的，所以用户在使用时，每次登录都需要输入密码，而 server 端不需要为这个用户创建账号。证书中心也是一台 Linux 主机，也需要生成自己的私钥，用于产生后续的证书中心的签名证书。这样就可以保证必须要有证书中心的私钥才能够给别人颁发签名，从而提高伪造证书的难度。

证书中心在生成了自己的私钥之后，需要生成可以给其他 server 签名的证书，代码如下。

```
openssl req -new -key private.key -out ca.csr //证书中心生成请求证书
openssl x509 -req -in ca.csr -extension v3_ca -signkey private.key -out
ca_sign.crt //证书中心给自己生成的请求证书签名，表明自己批准自己请求通过，并且生成一个可以给其他用户签名的签名证书给证书中心使用
```

这样证书中心的搭建就完成了。当一个 server 需要请求证书的时候，输入如下代码。

```
openssl genrsa -out private.key 2048//生成 server 自己的私钥，这里省略了加密  
openssl req -new -key private.key -out server.csr //server 生成请求证书
```

这时 server 生成了请求证书，可以发送给证书中心进行签名，代码如下。

```
openssl x509 -req -in server.csr -extensions v3_ca -CA ca_sign.crt -CAkey  
private.key -CAcreateserial -out sign.crt //证书中心为 server 最终生成签名过的  
证书，server 现在是被认证中心认证过的服务器了
```

这样就完成了 server 的证书请求。客户端只需要加载了 CA 公开的证书 ca\_sign.crt 就可以认证所有被这个证书中心签名过的 server 了。但由于颁发证书需要 CA 的 private.key，所以客户端拿到了证书也无法给别人颁发证书。

服务器也可能被伪造，所以客户端访问了假的服务器也是一场灾难。服务器一般像 CA 申请一个 csr 证书文件，这个证书文件是用 CA 的私钥加密过的。

在 PC 的浏览器中一般都预存了很多大型的 CA 的证书，这个 CA 的证书都是 CA 自签名的，里面记录了 CA 的公钥。当遇到持有某未在浏览器的信任 CA 列表中的 CA 签名的服务器 csr 证书文件的时候，客户端的浏览器一般会弹出对话框询问你是否相信这个 CA 签名的 csr 证书，如果你选择相信，也就意味着同时相信了这个服务器。已在信任列表的 CA，遇到这些 CA 签名的 csr 文件客户端会直接信任。

## 7.2.5 数字签名

还有一种认证叫作数字签名，这个认证方式就是典型的使用私钥来加密，使用公钥来解密的应用。这是非对称加密最有意思的一点，就是加密和解密的密钥可以互换。虽然只能公布质数积，但是不一定要用质数积来加密，也可以用它来解密。

## 7.2.6 密钥交换

常见的密钥交换方式有以下两种。

(1) 公钥加密实现：发送方用接收方的公钥加密自己的密钥，接收方用自己的

私钥解密得到发送方的密钥，反过来亦然，从而实现密钥交换。

(2) 使用 DH 算法：发送方和接收方前提协商使用同一个大素数  $P$  和生成数  $g$ ，各自产生随机数  $X$  和  $Y$ 。发送方将  $g$  的  $X$  次方  $\bmod P$  产生的数值发送给接收方，接收方将  $g$  的  $Y$  次方  $\bmod P$  产生的数值发送给发送方，发送方再对接收的结果做  $X$  次方运算，接收方对接收的结果做  $Y$  次方运算，最终形成密码，密钥交换完成。简单地说，DH 算法就是用来做对称加密的密钥生成交换的，之前这个工作是由非对称加密算法完成的，因此使用了这个算法，非对称加密算法就很大程度上失去了存在的意义。普通的 DH 算法被证明无法抵抗中间人攻击，因此目前与椭圆曲线结合，并且添加 AEAD 特性。

## 7.3 Linux 用户和权限系统

我们使用 Linux 系统，一定会使用一个用户账号，没有用户账号就不可能使用任何的系统功能，包括系统调用，因为系统调用本身也是要有用户的。我们刚登录一个系统，需要一个 login 程序，验证了用户名密码后，就会返回一个 Shell，后面执行的内容都是在 Shell 中执行的。我们也可以发现用户和密码都存储在 `/etc/passwd` 和 `/etc/shadow` 这些文件中，很有可能使用户产生权限校验是在系统级做的，而不是在内核中做的错觉。所有的权限校验都是在内核中完成的，但是策略都是在用户空间配置的。

### 7.3.1 系统启动时的权限

系统执行命令除了通过 Shell 启动，还可以通过其他的进程启动。由于 Linux 内核设计的进程继承关系，第一个进程是 `init` 进程，这个进程是 `root` 权限，拥有最高的完全权限。如果直接修改 `init` 程序，在里面实现逻辑，也是具有完全权限的，内核的所有权限检查都可以完全通过。

后来启动的进程全部是 `init` 进程复制出来的，Linux 有两种复制接口：`fork` (`vfork`) 和 `clone`。`fork` 就是完全复制，然后使用 `execve` 系统调用执行新的命令。`execve` 系统调用的参数也只是路径、参数和环境变量，而继承自父进程的所有权限和句柄都还在。`clone` 是可以指定复制内容的，例如文件句柄、根文件系统、命名空间等，但



是 `clone` 仍旧不能用其他的用户去执行程序，也不能设定更高权限的 `capabilities` 等权限。`vfork` 是因为 `fork` 调用过于占用内存（会拷贝整个内存空间），而 `vfork` 暂停父进程的执行，直接在父进程的地址控制中执行 `execve` 系列调用。这 3 个 API 的内部实际都是调用一个内核内部函数 `do_fork`。使用 `vfork` 的时候必须要注意，不能在子进程调用 `execve` 之前使用局部变量，否则会破坏父进程的栈，退出的时候要使用只清内核数据的 `_exit` 函数，而不能使用 `exit` 函数。

启动后面的低权限进程和用户的过程是一个不断降低权限的过程。例如不共享打开的文件句柄、设置新的更小的 `capabilities` 权限、使用权限更小的用户启动程序等。降权的方法是使用高权限的进程调用特定的系统调用，例如使用 `unshare`、`prctl` 或 `setuid` 来完成。这些进程权限高的原因是使用了权限高的用户。

`su` 命令是我们常用的切换到其他用户执行命令的方法，它所使用的就是 `setuid` 系统调用方法将当前执行的具有 `root` 权限的用户降级为非 `root` 用户，或者 `sudo -s` 从低权限用户切换到 `root` 用户。`unshare` 和 `prctl` 都不能用来切换用户，而是用来控制偏功能性的权限（`capabilities`、`namespace` 等）。`setuid` 系列调用有很多，各有区别，这里不进行讨论。

### 7.3.2 系统启动后的权限

系统正常运行后，进程要么由其他的进程启动，要么由 `Shell` 启动，由其他进程启动的很多也是由 `Shell` 启动的。`Shell` 是 `Linux` 系统的中坚力量，其本身也必须有一个所属用户。例如登录使用的用户 `user1`，那么 `login` 程序（`root` 权限）验证了这个用户名和密码后，就可以 `setuid` 设置启动的 `Shell` 以特定的用户权限运行。图形界面也是使用特定的用户运行了基础的组建之后再衍生的（例如 `lightdm`）。

`Shell` 之所以重要是因为它就是中转的中心。当然我们完全可以不需要 `Shell`，使用图形界面做中转中心，`Windows` 就是这么做的，也可以使用一个服务进程，接收远端的 `socket` 指令作为中转中心。`SSH` 是服务进程作为中转中心的例子，但是登录之后中心还是转变为了 `Shell`。虽然 `Python` 等脚本语言的能力已经非常强，但是在系统接口层面，只有 `Shell` 能够做到将程序、命令与字符串有机结合，使用起来不仅感觉它很强大而且非常简便。

还可以发现 `/etc/passwd` 中给每个用户都定义了 `Shell`，也可以使用 `/bin/false`，可

以看出 Shell 的思想已经固化到系统中了。所以 Shell 不再是一个简单的应用程序，而是一个系统组件（它本质就是一个应用程序）。

当我们在 Shell 中切换用户的时候，是用新的用户开启了新的 Shell。这个 Shell 的執行者就是新的用户，执行的命令也自然继承了新用户。而我们可以通过 `su` 或 `sudo` 改变这种情况。`su` 可以运行在低权限的 Shell 中，也可以切换到高权限的 Shell，因为 `su` 命令本身是有 `suid` 的，这个二进制标志可以让进程被普通用户执行，但是却拥有 `root` 的操作权限。

### 7.3.3 内核中的用户和权限模型

Linux 中为了定义权限，设计了一个对象模型。这个模型主要组件有 `object`、`subject`、`context`、`action`、`rule`。

- 常见的对象都是 `object`（`Tasks`、`Files/inodes`、`Sockets`、`Message queues`、`Shared memory segments`、`Semaphores`、`Keys`）。
- 当一个 `object` 作用在其他 `object` 的时候，发起方就是 `subject`，最常见的变成 `subject` 的 `object` 的是 `task`。
- 无论是 `object` 还是 `subject`，都有自己的 `context`。`context` 主要是一些归属于自身的权限存储（`credentials`）。
- 一个 `subject` 作用到另外一个 `object` 的动作叫作 `action`。
- 限制一个 `action` 权限叫作 `rule`。`rule` 包括 `MAC` 和 `DAC` 两类，常见的 `SELinux` 规则就是在这里工作的。

整个模型运作起来，给所有的对象都赋予了静态和动态的权限，就形成了完整的权限系统。

我们常见的用户和文件权限工作在 `context` 层次，即 `credentials`。目前有 7 种 `credentials`，要同时满足这些权限检查，若不满足其中一个检查权限就不能通过安全检查。比较常见的权限检查有以下 3 种。

- 传统的 `uid` 系列，设置在文件 `object context credentials` 层次，但是 `euid` 系列设置在 `subject context credentials` 层次，这些都是在文件本身保存的，除非不是物理的文件。
- `capabilities` 是 Linux 独创的系统。其将系统分成了几个部分，权限一个一个子系

统地设置和删除。这是进程 `subject context` 的内容。

- LSM 是大部分大规模的安全系统所基于的内核安全机制。通过在常用的路径上安装钩子实现，是完全独立于模型的机制。

当我们创建一个进程的时候，内核知道是谁创建的，并且会设置该进程 `task_struct` 的对应的域（`cred` 和 `real_cred`），里面有存储有效用户和实际用户的 `id`。内核知道并不是通过系统调用传递的，而是进程复制产生的。就是上一个进程的 `id`。如果上一个进程要用其他的 `id` 启动进程，它会首先 `setuid` 改变自己（例如 `su` 命令），然后再做系统调用。代码如下。

```
nohup su $user -s /bin/bash -c "(ulimit -c 10000;./app$1 $2&)"
```

所以内核本身也不知道进程的具体用户的名字和它对应的 `Shell` 和组。这些都存储在用户可见的文件系统，由用户维护。调用 `su` 命令后面加的用户名是在用户空间查到的对应的 `uid`，登录时验证密码也是 `login` 程序直接通过文件查询计算的结果，并不需要经过内核。

内核中并没有专门的数据库存储用户和组信息，而是依附在每个进程上，每个进程都可以有多种用户和组的设置。

## 7.3.4 Linux 安全体系

内核只管权限。如果文件的所有者是普通进程，但是却调用了 `reboot`，只要执行的时候有 `root` 权限，这个程序就能够执行成功（除非额外设置了系统调用限制或 `capabilities` 等其他安全维度）。所以内核默认依赖于谁（`uid`）在执行程序，而不是依赖执行的内容。

而大部分额外的安全系统（例如 `Apparmor`、`SELinux`、`seccomp`、`capabilities`、`ACL` 等），都是对默认最有效的、基于用户的安全体系的补充，大多数作用于执行的内容，而不是执行的用户，并且作用在不同的位置。

### 1. 用户和组的概念

用户和组是独立于文件和进程而存在的，它们被存储在 `/etc/` 下的几个文件中。

一个用户可以属于多个组，有一个主要组，其他的叫作 `supplementary group`，这些组之间没什么区别。文件 `object` 静态地包含了用户和组的设置，而 `task`（进程）



也分为静态和动态的用户和组的设置。

## 2. 文件 object 权限

文件有标准权限和扩展权限。标准权限包括用户、组、其他组 3 个的 r、w、x、s、t 权限。s 是让非 root 程序以 root 身份运行的标志位；t 是让进程启动后滞留内存不被回收的标志位，以及该文件属于哪个用户哪个组。

比较常见的扩展的有 i、a（主要是 ext 系列文件系统支持）。i 限制文件不能被删除；a 限制文件只能被追加。不同的文件系统可以实现不同的扩展权限，甚至可以用户自己定义，ACL 扩展是所有文件系统都有的能力。例如阻止 root 删除文件，如图 7-1 所示。

```
root@ubuntu:/# touch test1
root@ubuntu:/# chattr +i test1
root@ubuntu:/# rm test1
rm: cannot remove 'test1': Operation not permitted
root@ubuntu:/#
```

图 7-1

设置标准权限用 chmod，设置扩展权限用 chattr。还有其他的命令，如 getfacl、lsattr 命令。

## 3. 进程的 object 和 subject 权限

进程看起来是动态的，但是它们既是 object（静态的一面）也是 subject（动态的一面）。静态的一面就是我们设置在文件上的用户和组，以及其对应的权限。在大部分情况下，内核依据这个信息进行权限判断，这个用户叫作实际用户。

动态的一面就是当文件 suid 置位的时候，用户以 root 权限运行。这时进程的权限是 root 的，这个用户就是有效用户。理论上有效用户不一定是 root，但是目前都是这样，sgid 也是一个道理。

所以文件 object 本身的权限设置，既提供了文件 object 本身的权限，也提供了进程 object 的权限和进程 subject 的权限。

## 4. 总结

Linux 中默认是基于用户的安全，并且有各个文件系统的权限位补充配合。后续发展的多种维度的、基于行为的安全机制成为 Linux 安全系统发展的方向。安全系统大部分都在内核中完成，内核只是一个管理和验证系统，并不是所有的安全验证

都在内核中完成，系统层次可以完成相当多的权限验证。

## 7.4 网络安全

网络安全在 Linux 中处于极其重要的地位。网络安全几乎是 Netfilter 的天下，如今的 netfilter 演化出了 bpf 这种非常强大的功能，所以本书以面向未来的态度重点介绍 bpf。

### 7.4.1 netfilter 概览

最早的内核包过滤机制是 ipfwadm，后来是 ipchains，再后来是 iptables/netfilter 了，再往后就是现在的 nftables。不过 nftables 与 iptables 还处于争锋阶段，谁能胜出目前还没有定论，但是它们都属于 netfilter 项目的子成员。

#### 1. 钩子

netfilter 基于钩子，在内核网络协议栈的几个固定的位置有 netfilter 的钩子。我们知道数据包有两种流向，一种是给本机的：驱动接收——路由表——本机协议栈——上层应用程序——本机协议栈——路由表——驱动发送。另一种是要转发给别人的：驱动接收——路由表——转发——驱动发送。针对几个关键位置，netfilter 定义了以下几个钩子。

- NF\_IP\_PRE\_ROUTING 是在查路由表之前。
- NF\_IP\_LOCAL\_IN 是在查完路由表决定发送本机之后。
- NF\_IP\_FORWARD 是在查完路由表决定转发的时候。
- NF\_IP\_POST\_ROUTING 是要交给驱动发送之前。
- NF\_IP\_LOCAL\_OUT 是本机产生的数据交给驱动发送之前。

通过在这几个钩子位置注册函数，截断数据包的流动，可以完成数据包的过滤和转发功能。但转发功能一般只在路由器上打开，如果普通 PC 发现不是自己的数据包就会直接选择丢弃。所以，普通 PC 可以使用的钩子有 NF\_IP\_PRE\_ROUTING、NF\_IP\_LOCAL\_IN、NF\_IP\_LOCAL\_OUT、NF\_IP\_POST\_ROUTING 这 4 个。可以看到它们都是在 IP 层的钩子，然而这些钩子不仅可以处理 IP 层的数据，还可以拿

到完整的数据包，所以你想处理哪一层数据都是可以的。代码如下。

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
static struct nf_hook_ops ops; //钩子结构体
//数据包处理函数
unsigned int hook_func(unsigned int hooknum, struct sk_buff **skb,
const struct net_device *in, const struct net_device *out, int
(*okfn)(struct sk_buff *)) {
    return NF_DROP;          /* 丢弃所有包 */
}
int init_module()
{
    nfho.hook = hook_func;
    nfho.hooknum = NF_IP_PRE_ROUTING;
    nfho.pf = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST;
    nf_register_hook(&nfho); //注册钩子到内核
    return 0;
}
void cleanup_module()
{
    nf_unregister_hook(&nfho);
}
```

这样所有的数据包都会被钩子函数丢弃。

iptables 不是注册在钩子函数上的，但是位置都是一致的，是 netfilter 框架下的一个附属的功能，由 table、chain、rule 组成。

## 2. 用户空间使用 iptables、table、chain 和 rule

chain 和 rule 是 iptables 自创的概念，我们知道在钩子函数的地方可以执行指定的函数调用。iptables 系统就默认实现了几个调用，并且用统一的数据结构来组织这个调用的形式，这个组织结构就是 table、chain 和 rule。

在任何一个 hook 点都可以定义多个 table，一个 table 有多个 chain，每个 chain



中可以定义多个 rule。要记住的是 table 和 chain 只是容器，里面的 rule 才是真正发挥作用的规则。理论上我们可以在任何一个 hook 点进行过滤、nat、修改数据包等操作，但是 iptables 为了统一架构起见，在各个 hook 点定义了顺序的几个 table，每个 table 用来完成一类的工作。预定义的 table 包括 filter、nat 和 mangle。每一个 table 表示的是功能，并不是表示位置，一个 table 内部有多个 chain，其中每个 chain 位于特定的位置。

如图 7-2 所示的是一个内部已经定义的 table、chain 关系图表 (<https://upload.wikimedia.org/wikipedia/commons/3/37/Netfilter-packet-flow.svg>)。

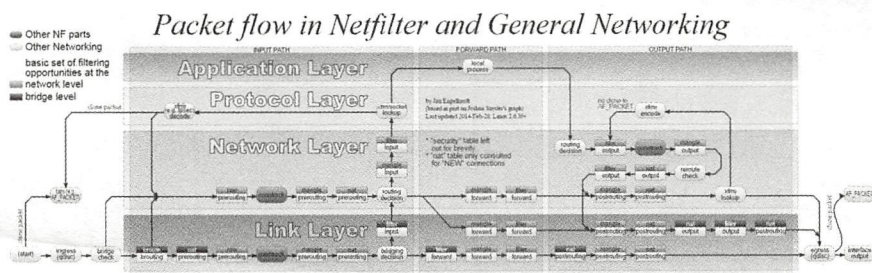


图 7-2

可以看到，预定义的 3 个表表示 3 种不同的功能，每个表都在一些 hook 点定义了一组 chain。当一个用户想要在某一个 hook 点做某一件事情时，就可以定位到 table（功能）——chain（位置）——rule（行为）来完成数据包的操作。

每一条 rule 的格式都是相同的，包括源 IP 地址、目的 IP 地址、上层协议、接口、操作（target）。但并不是每个域对于每个 chain 都是可用的，例如在 INPUT 的地方匹配输出接口就是永远匹配不到的，所以有效的 rule 在不同的 chain 上是不同的。

iptables 是个可扩展的软件，其对 TCP、UDP 等常用协议的支持都是通过扩展，iptables 本体只支持到 IP 层，只要使用对应的选项就会自动使用扩展。还有一些不是协议的扩展，这些扩展一般通过 iptables -m 调用，例如 iptables -m mac 可以用来匹配 MAC 地址。-m limit 可以用来限制每秒匹配的数目，超过数目的就放行。这些扩展包括但不限于以下几项。

- xt\_mac.ko: 匹配 MAC 地址。
- xt\_limit.ko: 限制每秒匹配的数目，超过数目的就放行。
- xt\_owner.ko: 用来匹配某个进程或用户创建的数据包。

- `xt_state.ko`: 用来匹配处于某个连接状态的数据包(例如 NEW、ESTABLISHED、RELATED)。
  - `xt_pkttype.ko`: 根据多播、广播还是单播来匹配包。
  - `xt_quota.ko`: 可以为一个 rule 设置 quota, 当 quota 达到后, 改 rule 失效。
  - `xt_recent.ko`: 允许你设置一个 IP 列表, 后续的 IP 列表的用户都不生效。
  - `t_string.ko`: 允许你匹配数据包中的一个字符串。
  - `xt_time.ko`: 允许你根据数据包的到达和离去时间进行匹配。
  - `xt_u32.ko`: 通过匹配检查数据包的某 4 位是否与要求的一致来进行操作。
- 下面的例子是用来做临时访问限制的语句, 代码如下。

```
iptables -t raw -A OUTPUT -d 1.2.3.4/32 -p tcp -m tcp --dport 80 --tcp-flags
SYN SYN -m connlimit --connlimit-above 2 --connlimit-mask 32
--connlimit-saddr -c 0 0 -j DROP
```

还有很多 target 的扩展、conntrack 的扩展和 IPv6 的扩展, 可以根据应用的类型进行匹配, 可以修改 TTL、TOS 等数据位, 基本能用得上的功能都有对应的扩展。

操作(target)也是可以扩展的, 常见的默认操作有 ACCEPT 和 DROP, 扩展的还有 LOG、REJECT 等, 用户还可以自己实现。还有两种默认的操作是 QUEUE 和 RETURN。RETURN 实现了各个规则之间的函数式调用; QUEUE 实现了数据包的排队。这些操作也都对应着具体的模块, 例如 `nft_queue.ko`、`nft_reject.ko`、`xt_LOG.ko`、`nft_log.ko` 等。

### 3. bpf

用户空间不仅可以添加规则, 还可以添加代码, 并通过 `xt_bpf` 模块实现。`iptables -m bpf -bytecode` 后面跟具体的 code 就好了, 从汇编编译成 code 的程序存在于 Linux 内核的 `tool/net` 目录下。

## 7.4.2 Filter (LSF、BPF、eBPF)

LSF (Linux socket filter) 起源于 BPF (Berkeley Packet Filter), 两者基础架构一致, 但 LSF 使用更简单。LSF 内部的 BPF 最早是 cBPF (classic), 后来 x86 平台首先切换到 eBPF (extended), 但由于很多上层应用程序仍然使用 cBPF (tcpdump、

iptables), eBPF 还没有支持很多平台, 所以内核提供了从 cBPF 向 eBPF 转换的逻辑, 并且 eBPF 在设计的时候也沿用了 many cBPF 的指令编码。但是在指令集合寄存器, 在架构设计上有很大不同 (例如 eBPF 已经可以调用 C 函数, 并且可以跳转到另外的 eBPF 程序)。

但是新的 eBPF 一出来就被“玩坏了”, 人们很快发现了它在内核 trace 方面的意义, 它可以保证绝对安全地获取内核执行信息 (虽然早期的版本出现了一些安全漏洞), 是内核调试和开发者的不二选择, 所以针对这个方面, 例如 kprobe、ktp、perf eBPF 等优秀的工作逐渐产生, 反而关注数据包过滤领域的人不多。tc (traffic controll) 是使用 eBPF 的一款优秀的用户端程序, 它允许不用重新编译模块就可以动态添加删除新的流量控制算法。netfilter 的 xtable 模块配合 xt\_bpf 模块, 就可以将 eBPF 程序添加到 hook 点来实现过滤。当然, 内核中提供了从 cBPF 到 eBPF 编译的函数, 所以在任何情况下想要使用 cBPF 都可以, 内核会自动检测和编译。

## 1. bpf 概览

核心原理是对用户提供了两种 socket 选项, 即 SO\_ATTACH\_FILTER 和 SO\_ATTACH\_BPF。允许用户在某个 socket 上添加一个自定义的 filter, 只有满足该 filter 指定条件的数据包才会发送到用户空间。因为 socket 有很多种, 你可以在各个维度的 socket 添加这种 filter, 如果添加在 raw socket, 就可以实现基于全部 IP 数据包的过滤 (tcpdump 就是这个原理)。如果你想做一个 http 分析工具, 就可以在基于 80 端口 (或其他 http 监听端口) 的 socket 添加 filter。还有一种使用方式是离线式的, 使用 libpcap 抓包存储在本地, 然后可以使用 bpf 代码对数据包进行离线分析, 这对于实验新的规则和测试 bpf 程序非常有帮助: SO\_ATTACH\_FILTER 插入的是 cBPF 代码, SO\_ATTACH\_BPF 插入的是 eBPF 代码。eBPF 是对 cBPF 的增强, 目前用户端的 tcpdump 等程序还是用的 cBPF 版本, 其加载到内核中后会被内核自动转变为 eBPF。代码如下。

```
echo 2 > /proc/sys/net/core/bpf_jit_enable
```

通过上述命令写入 0/1/2 可以实现关闭、打开、调试日志等 bpf 模式。

在用户空间处理数据包最简单、最通用的办法是使用 libpcap 的引擎。由于 bpf 是一种汇编类型的语言, 自己写难度比较高, 所以 libpcap 提供了一些上层封装可以直接调用。然而 libpcap 并不能提供所有需求, 比如 bpf 模块开发者的测试需求, 还

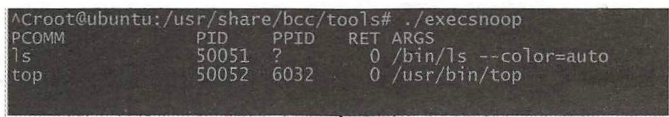


有高端的自定义 bpf 脚本的需求。这种情况下就需要自己编写 bpf 代码, 然后使用内核 tools/net/目录下的工具进行编译, 编译成 bpf 汇编代码, 再使用 socket 接口传入这些代码即可。bpf 引擎在内核中实现, 但是 bpf 程序的工作地点很多都需要额外的模块来支持, 常用的有 netfilter 自带的 xtable、xt\_bpf、可以实现在 netfilter 的 hook 点执行 bpf 程序、cls\_bpf 和 act\_bpf 可以实现对流量进行分类和丢弃 (qos)。

使用 eBPF 最好的方法是 BCC 工具集。了解其原理可以从 tc-bpf 入手, 在 ubuntu 16 中只需要简单的几步就能够安装好 BCC 工具集。

```
echo "deb [trusted=yes] https://repo.iovisor.org/apt/xenial
xenial-nightly main" | sudo tee /etc/apt/sources.list.d/iovisor.list
sudo apt-get update
sudo apt-get install bcc-tools libbcc-examples
```

如图 7-3 所示是一个使用 BCC 实现命令行记录器的例子 (在 bash 内建命令不会被抓取)。如果你不是一个内核开发者, 而是一个高级的内核使用者, 正确的方式应该以目的为导向, 例如 BPF 几乎能做到传统方式可以做到的任何事情, 并且只需要使用 BCC 的一系列封装即可, 此时我们就没必要自己过于深入地研究 BCC 到底是怎么运行的。



```
ACroot@ubuntu:/usr/share/bcc/tools# ./execsnoop
PCOMM      PID    PPID    RET  ARGS
ls          50051   ?       0    /bin/ls --color=auto
top         50052  6032    0    /usr/bin/top
```

图 7-3

内核对 bpf 的完整支持是从 3.9 版本开始的, 作为 iptables 的一部分存在, 默认使用的是 xt\_bpf, 用户端的库是 libxt\_bpf。iptables 一开始对规则的管理方式是顺序地一条条执行, 这种执行方式难免在匹配数目多的时候产生性能瓶颈。添加了 bpf 支持后, 灵活性就大大提升了。以上所有提到的可以使用 bpf 的地方均指同时可使用 eBPF 和 cBPF, 因为内核在执行前会自动检查是否需要转换编码。

内核的 bpf 支持是一种基础架构, 只是一种中间代码的表达方式, 是向用户空间提供一个向内核注入可执行代码的公共接口。只是目前的大部分应用是使用这个接口来做包过滤。其他的, 如 seccomp BPF 可以用来实现限制用户进程可使用的系统调用; cls\_bpf 可以用来将流量分类; PTP dissector/classifier 等都是使用内核的 eBPF 语言架构来实现各自的目的, 并不一定是包过滤功能。

bpf 的一些常用的工具有 clang、tcpdump、tools/net、seccomp BPF、IO visitor、ktap 和 BCC。

## 2. cBPF 汇编架构分析

首先必须表明 bpf 的架构仍在不断演变，这里介绍的是早期版本的原理，未来 bpf 的发展可能会有比较大的变动。下面介绍 cBPF 的逻辑，cBPF 通常用于用户端 bpf 编程。

cBPF 中每一条汇编指令都是如下所示的格式。

```
struct sock_filter {
    __u16  code; /* 功能代码 */
    __u8   jt; /* Jump true */
    __u8   jf; /* Jump false */
    __u32  k;   /* 通用的多用途域 */
};
```

code 是真实的汇编指令；jt 是指令结果为 true 的跳转；jf 是为 false 的跳转；k 是指令的参数，根据指令的不同而不同。一个 bpf 程序编译后就是一个 sock\_filter 的数组，可以使用类似汇编的语法进行编程，然后使用内核提供的 bpf\_asm 程序进行编译。

bpf 在内核中实际上是一个虚拟机，有自己定义的虚拟寄存器组，和我们熟悉的 Java 虚拟机的原理一致。这个虚拟机的设计是 LSF 的成功所在。cBPF 有 3 种寄存器，如表 7-1 所示。

表 7-1

寄存器	位数	说 明
A	32	所有加载指令的目的地址和所有指令运算结果的存储地址
X	32	二元指令计算 A 中参数的辅助寄存器（例如移位的位数、除法的除数）
M[0-15]	32	可以自由使用的 16 个寄存器

我们最常见的用法莫过于从数据包中取某个字的数据来做判断。按照 bpf 的规定，我们可以使用偏移来指定数据包的任何位置，而很多协议是很常用的并且是固定的，例如端口和 IP 地址等，bpf 就为我们提供了一些预定义的变量，只要使用这个变量就可以直接取值到对应的数据包位置，如表 7-2 所示。如果读取数据包时 eBPF 程序想读取超过数据包长度的内容，eBPF 程序将会被停止执行。

表 7-2

bpf 变量	数据包域
len	skb->len
proto	skb->protocol
type	skb->pkt_type
poff	负载偏移
ifidx	skb->dev->ifindex
nla	Netlink 属性偏移 ( type X, 偏移 A )
nlan	Nestted 的 Netlink 属性偏移 ( type X, 偏移 A )
mark	skb->mark
queue	skb->queue_mapping
hatype	skb->dev->type
Rxhash	skb->hash
cpu	raw_smp_processor_id()
vlan_tci	skb_vlan_tag_get(skb)
vlan_avail	skb_vlan_tag_present(skb)
vlan_tpid	skb->vlan_proto
Rand	prandom_u32()

更可贵的是这个列表还可以由用户自己去扩展, 各种 bpf 引擎的具体实现还会定义各自的扩展。

### 3. eBPF 汇编架构分析

用户可以提交 cBPF 的代码, 首先将用户提交的结构体数组进行编译, 编译成 eBPF 代码 (如果提交的代码是 eBPF 就不用编译了), 然后将 eBPF 代码转变为可直接执行的二进制。cBPF 在很多平台内核还在使用, 这个代码就和用户空间使用的那种汇编是一样的, 但是在 x86 的架构中, 现在内核态已经都切换到使用 eBPF 作为中间语言了, 也就是说 x86 在用户空间使用的汇编和在内核空间使用的并不一样。但是内核在定义 eBPF 的时候已经尽量复用 cBPF 的编码了, 有的指令的编码和意义, 如 BPF\_LD 都是完全一样的。然而在还不支持 eBPF 的平台中, cBPF 则是唯一可以直接执行的代码, 不需要转换为 eBPF。

eBPF 对每一个 bpf 语句的表达与 cBPF 稍有不同, 定义如下:

```
struct bpf_insn {
    __u8    code;          /* 功能代码 */

```



```

__u8   dst_reg:4;    /* 目标寄存器 */
__u8   src_reg:4;    /* 源寄存器 */
__s16  off;          /* 带符号的偏移 */
__s32  imm;          /* 带符号的立即数常数 */
};

```

其寄存器也不同，如表 7-3 所示。

表 7-3

寄存器	介 绍	x86 物理寄存器映射
R0	调用内核函数的返回值、eBPF 的退出值	rax
R1 - R5	eBPF 调用内核函数的参数	R1 - rdi R2 - rsi R3 - rdx R4 - rcx R5 - r8
R6 - R9	由于被调用者的内核函数会进行入栈操作，这些寄存器就是入栈保存的寄存器	R6 - rbx R7 - r13 R8 - r14 R9 - r15
R10	只读的栈寄存器	rbp

为了配合更强大的功能，eBPF 汇编架构使用的寄存器数目有所增加，上述寄存器的存在，充分体现了函数调用的概念，而不再是加载处理的原始逻辑。有了函数调用的逻辑设置可以直接调用内核内部的函数（这是一个安全隐患，但是内部有规避机制）。不但如此，由于这种寄存器架构与 x86 等 CPU 的真实寄存器架构非常像，实际的实现正是采用了直接的寄存器映射，也就是说这些虚拟的寄存器实际上是使用的是同功能的真实寄存器，这无疑是对效率的极大提高。而且在 64 位操作系统的计算机上，这些计算机将会有 64 位的宽度，完美地发挥硬件能力。虽然目前的 64 位操作系统的支持还不太完善，但已经可以用了。

因为 x86-64 规定 rdi、rsi、rdx、rcx、r8、r9 用来做参数传递；rbx、r12~r15 用来做调用者保留，所以有了如上的映射方式。R1~R5 是函数调用的参数寄存器，每次调用了之后这几个寄存器的值可能被改变，所以需要每次调用其他函数后都要重新填充。这 5 个寄存器被映射到对应平台的实际参数寄存器中，在 x86-64 下，寄存器的映射方法如下：R6~R9 会在 eBPF 调用其他函数前后保持一致，所以可以用来放 eBPF 变量。

目前的内核实现，只能在 eBPF 程序中调用预先定义好的内核函数，不可以调用其他的 eBPF 程序（但是可以通过 map 的支持跳转到其他 eBPF 程序，然后再跳回来）。这看起来无关紧要，但是却是一个很好的能力，这就意味着你可以使用 C 语言来实现 eBPF 程序逻辑，eBPF 只需要调用这个 C 函数就好了。

#### 4. eBPF 的数据交互：map

eBPF 是程序，但它还可以访问外部的数据，重要的是这个外部的数据可以在用户空间中管理。这个 k-v 格式的 map 数据体是通过在用户空间调用 bpf 系统调用来实现创建、添加、删除等操作管理的。

用户可以同时定义多个 map，使用 fd 来访问某个 map。有一个特殊种类的 map 叫作 program array。这个 map 存储的是其他 eBPF 程序的 fd，通过这个 map 可以实现 eBPF 之间的跳转，跳转走了就不会跳转回来，最大深度是 32，这样防止无限循环递归的产生（也就是可以使用这个机制实现有限递归）。更重要的是，这个 map 在运行时可以通过 bpf 系统调用动态的改变，这就提供了强大的动态编程能力。比如可以实现一个大型过程函数的中间某个过程的改变。实际上一共有 3 种 map，如下所示。

```
BPF_MAP_TYPE_HASH,           //hash 类型
BPF_MAP_TYPE_ARRAY,          //数组类型
BPF_MAP_TYPE_PROG_ARRAY,      //程序表类型
```

#### 5. eBPF 的直接编程方法

除了在用户空间通过 nettable 和 tcpdump 来使用 bpf，在内核中或者在其他通用的编程中可以直接使用 C 语言编写 eBPF 代码，但是需要 LLVM 支持。下面举个例子详细说明，LLVM 对 bpf 编程的封装支持，如图 7-4 所示。

在用户空间通过使用 bpf 系统调用的 BPF\_PROG\_LOAD 方法，就可以发送 eBPF 的代码进内核，发送的代码不需要再做转换，因为其本身就是 eBPF 格式的。如果要在内核空间模块使用 eBPF，可以直接使用对应的函数接口插入 eBPF 程序到 sk\_buff，提供强大的过滤能力。

Linux 提供的系统调用 bpf，用于操作 eBPF 相关的内核部分，代码如下。

```
#include <linux/bpf.h>
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

```

/* helper functions called from eBPF programs written in C */
static void *(*bpf_map_lookup_elem)(void *map, void *key) =
    (void *) BPF_FUNC_map_lookup_elem;
static int (*bpf_map_update_elem)(void *map, void *key, void *value,
    unsigned long long flags) =
    (void *) BPF_FUNC_map_update_elem;
static int (*bpf_map_delete_elem)(void *map, void *key) =
    (void *) BPF_FUNC_map_delete_elem;
static int (*bpf_probe_read)(void *dst, int size, void *unsafe_ptr) =
    (void *) BPF_FUNC_probe_read;
static unsigned long long (*bpf_ktime_get_ns)(void) =
    (void *) BPF_FUNC_ktime_get_ns;
static int (*bpf_trace_printk)(const char *fmt, int fmt_size, ...) =
    (void *) BPF_FUNC_trace_printk;
static void (*bpf_tail_call)(void *ctx, void *map, int index) =
    (void *) BPF_FUNC_tail_call;
static unsigned long long (*bpf_get_smp_processor_id)(void) =
    (void *) BPF_FUNC_get_smp_processor_id;
static unsigned long long (*bpf_get_current_pid_tgid)(void) =
    (void *) BPF_FUNC_get_current_pid_tgid;
static unsigned long long (*bpf_get_current_uid_gid)(void) =
    (void *) BPF_FUNC_get_current_uid_gid;
static int (*bpf_get_current_comm)(void *buf, int buf_size) =
    (void *) BPF_FUNC_get_current_comm;

/* llvm builtin functions that eBPF C program may use to
 * emit BPF_LD_ABS and BPF_LD_IND instructions
 */
struct sk_buff;
unsigned long long load_byte(void *skb,
    unsigned long long off) asm("llvm.bpf.load.byte");
unsigned long long load_half(void *skb,
    unsigned long long off) asm("llvm.bpf.load.half");
unsigned long long load_word(void *skb,
    unsigned long long off) asm("llvm.bpf.load.word");

```

图 7-4

这个函数的第一个参数 `cmd` 就是内核支持的操作种类，包括 `BPF_MAP_CREATE`、`BPF_MAP_LOOKUP_ELEM`、`BPF_MAP_UPDATE_ELEM`、`BPF_MAP_DELETE_ELEM`、`BPF_MAP_GET_NEXT_KEY`、`BPF_PROG_LOAD` 这 6 种。从名字上就可以发现，前面 5 种是用来操作 `map` 的，这个 `map` 前面说过，是用户程序和内核 eBPF 程序通信的唯一方式。这 5 个调用类型都是给用户空间的程序使用的，最后一个 `BPF_PROG_LOAD` 方式用来向内核中加载 eBPF 代码体。

第二个参数 `attr` 是 `cmd` 参数的具体参数，根据 `cmd` 的不同而不同，如果是 `load` 的话还包括了完整的 eBPF 程序。

值得注意的是，每一个 `map` 和 eBPF 都是一个文件，都有对应的 `fd`。这个 `fd` 在用户空间看来与其他 `fd` 无异，可以释放，可以通过 UNIX Domain Socket 在进程间传递。如果定义一个 `raw` 类型的 `socket`，在其上附上 eBPF 程序过滤程序，甚至就可以直接充当 `iptables` 的规则使用。

目前内核中已经有很多与 `bpf` 相关的功能模块，例如 `act_bpf`、`cls_bpf`、`xtable`、`xt_bpf`。IO visitor 可能是基于 eBPF 相关的最大型的系统了，由多个厂商参与开发。



## 6. 业界对 eBPF 的 tracing 使用

我们知道 eBPF 有 map 数据结构，有程序执行能力，这就是完美的跟踪框架。比如通过 kprobe 将一个 eBPF 程序插入 I/O 代码，监控 I/O 次数，然后通过 map 向用户空间汇报具体的值。用户端只需要每次使用 bpf 系统调用查看这个 map 就可以得到想要统计的内容了。那么为什么要使用 eBPF，而不是直接使用 kprobe 的 C 代码本身呢？这就是 eBPF 的安全性，其机制设计使其永远不会 crash 掉内核，不会与正常的内核逻辑发生交叉影响。可以说通过选择工具避免了可能发生的很多问题。更可贵的是 eBPF 是原生地支持 tracepoint，这就为 kprobe 不稳定的情况提供了可用性。

在 Brendan Gregg 的博客中有一个使用 eBPF 进行 kprobe 测试的例子。ktap 创造性地使用 eBPF 机制实现了内核模块的脚本化。使用 ktap，可以直接使用脚本编程，无须编译内核模块，就可以实现内核代码的追踪和插入。这背后就是 eBPF 和内核的 tracing 子系统。此外华为也在为 bpf 添加 perf 脚本的支持能力。

eBPF 起源于包过滤，目前在 trace 市场得到越来越广泛的应用。

## 7. 意义和总结

目前使用传统的 bpf 语法和寄存器在用户空间写 bpf 代码，代码在内核中会被编译成 eBPF 代码，然后编译为二进制被执行。传统的 bpf 语法和寄存器简单，更面向业务，类似于高层次的编程语言，而内核的 eBPF 语法和寄存器复杂，类似于真实的汇编代码。

那么为何内核要大费周章地实现如此一个引擎呢？因为轻量级、安全性和可移植性。由于是中间代码，可移植性不必说，但是使用内核模块调用内核的函数接口一般也是可移植的，所以这个并不是很重要的理由。eBPF 代码在执行的过程中被严格地限制了禁止循环和安全审查，使得 eBPF 被严格地定位于提供过程式的执行语句块，甚至连函数都算不上，最大不超过 4096 个指令。所以这就是其定位：轻量级、安全、不循环。

## 8. eBPF 编程

一个 eBPF 函数被调用的时候会自动带一个参数 ctx 传递给 eBPF 程序，放在 R1 里（在 `__bpf_prog_run()` 函数中实现）。这个参数 ctx 对于用作 filter 的 eBPF 程序来说是 skb；对于用作 seccomp 来说是 seccomp\_data。所以可以看出，一个使用 `xt_bpf`

模块的 eBPF 过滤程序的原理是在约定的 hook 点, eBPF 被调用, skb 被作为第一个参数传进 eBPF 程序, 执行完后, 返回值 R0 作为判断这个包处理结果的返回值(是否丢弃等)。

在指令编码上, 使用 8 位进行编码 (code), 针对不同的指令, 这 8 位指令编码的使用情况是不同的, 但 LSB 的最后 3 位指令编码都是用来存储指令类型的。被存储的指令类型主要有以下几种, 如表 7-4 所示。

表 7-4

指令类型	指令编码
BPF_LD	0x00
BPF_LDX	0x01
BPF_ST	0x02
BPF_STX	0x03
BPF_ALU	0x04
BPF_JMP	0x05
BPF_ALU64	0x07

以上是表示指令编码的后 3 位。BPF\_JMP 是跳转类型的指令, 目前有 10 个。BPF\_ALU 和 BPF\_ALU64 是运算类指令, 目前有 14 个。而剩下的则是加载与存储类型的指令。也就是说 eBPF 一共有 3 大类指令: 跳转、运算、加载与存储。指令类型指代了一大类的指令, 由于指令是一个字节的编码, 指令类型只占了 3 个位。其他 5 位的定义与具体的指令类型有关。例如当最后 3 位是 BPF\_ALU 或 BPF\_JMP 时, 8 位的指令编码如表 7-4 所示, 中间一位有两种取值, 表示这个指令使用的是源寄存器, 如表 7-5 所示。

表 7-5

中间位	编码	cBPF	eBPF
BPF_K	0x00	X 寄存器作为源操作数	用 src_reg 作为源操作数
BPF_X	0x08	用 32 位的立即数作为源操作数	用立即数作为源操作数

所以一个指令分为 3 部分, 这 3 部分分别是一个字节的不同位, 这些位之间的组合通过“或”或者是“加”的方式进行操作。

- BPF\_XOR | BPF\_K | BPF\_ALU: 意味着  $\text{src\_reg} = \text{src\_reg} \wedge \text{imm32}$ 。
- BPF\_MOV | BPF\_X | BPF\_ALU: 将 src\_reg 的值移动到 dst\_reg。

- BPF\_ADD | BPF\_X | BPF\_ALU64:  $\text{dst\_reg} = \text{dst\_reg} + \text{src\_reg}$ 。

加载与存储指令也有多个，每个指令可以操作的数据的大小是不一样的，这个数据的大小的区别在于中间两个位，如表 7-6 所示。

表 7-6

中间 2 位	编 码	作用：操作数据字节的个数
BPF_W	0x00	4 个字节
BPF_H	0x08	2 个字节
BPF_B	0x10	1 个字节
BPF_DW	0x18	8 个字节

前 3 个位的编码表示操作模式编码，这些编码的用处如表 7-7 所示。

表 7-7

宏	编 码	用 处
BPF_IMM	0x00	立即数的移动
BPF_ABS	0x20	数据包固定偏移
BPF_IND	0x40	数据包可变偏移
BPF_MEM	0x60	内存偏移的值
BPF_LEN	0x80	数据包大小，只在 cBPF 中有效
BPF_MSH	0xa0	数据包 ip 头的大小，只在 cBPF 中有效
BPF_XADD	0xc0	异或

其中 BPF\_ABS 和 BPF\_IND 只能用在数据包处理上，这时 R6 里面是输入数据包 sk\_buff，R0 是输出数据包。在很多情况下并不需要中间位和模式位，3 位的类型位就能完成很多的工作。下面举一个例子，bpf 程序代码如下。

```

bpf_mov R6, R1 /*保存上下文 */
bpf_mov R2, 2
  bpf_mov R3, 3
  bpf_mov R4, 4
  bpf_mov R5, 5
bpf_call foo
bpf_mov R7, R0 /* 保存 foo() 返回值 */
bpf_mov R1, R6 /* 恢复上下文*/
bpf_mov R2, 6

```



```

    bpf_mov R3, 7
    bpf_mov R4, 8
    bpf_mov R5, 9
    bpf_call bar
    bpf_add R0, R7
bpf_exit

```

在这个程序中用到的指令的编码如下。

```

#define BPF_MOV 0xb0
#define BPF_ADD 0x00
#define BPF_CALL 0x80
#define BPF_EXIT 0x90

```

这个程序用 C 语言表达如下。

```

u64 bpf_filter(u64 ctx)
{
    return foo(ctx, 2, 3, 4, 5) + bar(ctx, 6, 7, 8, 9);
}

```

我们将这个 bpf 程序编译（注意这里明显直接使用使用的是 eBPF）如下。

```
bpf_asm test.bpf
```

然后把得到的 bytecode 使用 iptables 注入到内核，如下所示。

```

iptables -A INPUT -p udp --dport 1024 -m bpf --bytecode "14,0 0 0 20,177
0 0 0,12 0 0 0,7 0 0 0,64 0 00,21 0 7 124090465,64 0 0 4,21 0 5 1836084325,64
0 0 8,21 0 3 56848237,80 0 012,21 0 1 0,6 0 0 1,6 0 0 0," -j ACCEPT

```

可以用的编译器有 llvm、内核提供的编译器、iovisor 的 uBPF 编译器。BPF CompilerCollection (BCC) 这个工具集包含很多用来观测内核性能的工具，全部使用 eBPF，并且提供了 Python 的外部编程能力。其也是使用 llvm 用作底层编译器，并且整合了 llvm 中对 bpf 支持的最新进展，但是要求内核支持版本是 4.1。使用 llvm 可以使用如下命令编译。

```
clang-3.7 -O2 -target bpf -c sockex1_kern.c -o soc1.o
```

这样可以编译一个包含了 map 和 eBPF 代码的 ELF 文件。然而这个文件并不是用来直接插入内核的 eBPF 程序代码，只是一个包含了需要插入内核的各种信息的

一个集合体，内核代码在 `sample/bpf` 里面有提供解析这个文件的代码逻辑，可以自动实现解析和插入内核。这都是使用的 `bpf` 系统调用进行插入的 `eBPF` 代码，然而这个系统调用只支持插入到特定的内核位置，代码如下。

```
BPF_PROG_TYPE_SOCKET_FILTER,    //附在某一个 socket 上，只对某一个 socket 产生影响
BPF_PROG_TYPE_KPROBE,           //附在 kprobe 上
BPF_PROG_TYPE_SCHED_CLS,        //附在 cls_bpf 分类模块上
BPF_PROG_TYPE_SCHED_ACT,        //附在 act_bpf 模块上
```

可以看到，程序代码无法插入到我们希望的 `hook` 点上（因为 `hook` 点是 `netfilter` 的基础设施，`netfilter` 使用 `xt_bpf` 来支持 `bpf`，所以就没有在系统调用层级支持）。而类型 `BPF_PROG_TYPE_SOCKET_FILTER` 的 `bpf`，即使是使用 `raw`，得到的数据也只是一份拷贝。这就注定了这个机制只能用来做分析，不能用来做过滤。所以目前这条路行不通，过滤只能采用 `xt_bpf`。

`xt_bpf` 是 `iptables` 的扩展，可以使用 `-m` 参数传递 `bpf` 代码进内核。我们可以使用内核提供的 `bpf` 编译程序编译代码，或者是 `tcpdump -ddd`，或者是 `iptables` 的 `nfbpf_compile.c` 程序。但是这 3 个编译得到的都是 `cBPF` 代码。`iptables` 的用户端程序也只支持 `cBPF` 代码。所以使用 `xt_bpf` 模块的合理选择应该是使用 `cBPF` 编程。如果要使用 `eBPF` 编程，可行的方法是在自己实现钩子函数中执行 `eBPF`，或者是复用 `xt_bpf` 的代码。

## 7.5 函数调用的调试

函数调用分为内核函数调用和库函数调用，还有二进制文件本身的函数调用。想要对函数调用进行测试，一般需要用户端的 `strace` 命令行工具，或者 `audit` 审计系统。

原理上 `ptrace` 可以在用户层拦截和修改用户进程的系统调用。在执行系统调用之前，内核会先检查当前进程是否处于被“跟踪”（`Ptrace` 的步进）的状态。如果是的话，内核暂停当前进程并将控制权交给跟踪进程，使跟踪进程得以察看或者修改被跟踪进程的寄存器，这种机制是使用内核的系统调用配合做到的。

有一种基于 `trap` 的进程追踪方式，在打算中断的二进制文件位置上插入陷阱指

令 (int 3)，然后程序会调用自定义的陷阱代码，比较知名的工具是 `substrate hook`，逆向中很常用，这是由内核和 CPU 的机制共同完成的。

有一种方案是使用 `systemtap`。`systemtap` 本身是利用内核 `kprobe` 在内核事件中插入中间代码，是完全基于内核机制的。

有一种方案是 `valgrind` 采用的中间代码，它先把二进制文件反汇编，然后再编译为中间代码，这个重新编译的过程就可以自由插入自己的逻辑代码了。其他类似的还有 Android 的 `smali`，`llvm` 的 `IR`。

还有一种使用跳转表格的做法，就是在二进制文件中添加额外的函数，将原来的函数直接通过修改二进制的方式替换为我们自己函数表的调用。而我们会重新实现原来的函数（也可以直接拷贝），这种做法速度快，常用的软件是 `Dyninst`。还可以先约定函数命名，然后声明为 `weak` 符号，这样在链接的时候就可以链接另外一个强符号来动态地改变函数逻辑了。这种方式常用于用户程序对库函数的覆盖实现。

## 7.6 内核调试

内核也是一个程序，调试程序常用的方法有 3 种，即打印信息、断点执行和插入探测点。

### 1. `printk`

最常用的内核打印输出方式是 `printk` 函数，可以修改内核代码里任何想要打印的信息。健壮性是 `printk` 最大的特性。几乎在任何地方、任何时候内核都可以调用它（中断上下文、进程上下文、持有锁时、多处理器处理时等）。在系统启动过程中，终端初始化之前，在某些地方是不能调用的。如果真的需要调试系统启动过程最开始的地方，有以下两种方法可以使用。

- 使用串口调试，将调试信息输出到终端设备上。
- 使用 `early_printk()`，该函数在系统启动初期就有打印能力，但它只支持部分硬件体系。

`printk` 和 `printf` 主要的区别就是前者可以指定一个 LOG 等级，内核根据这个等级来判断是否在终端上打印消息，内核把比指定等级高的所有消息显示在终端。可以使用下面的方式指定一个 LOG 级别，代码如下。



```
printk(KERN_CRIT "Hello, world!\n");
```

第一个参数并不是一个真正的参数,因为其中没有用于分隔级别(KERN\_CRIT)和格式字符的逗号(,)。KERN\_CRIT 本身只是一个普通的字符串(表示的是字符串"<2>")。作为预处理程序的一部分,C 语言会将这两个字符串组合在一起。组合的结果是将日志级别和用户指定的格式字符串包含在一个字符串中。

内核使用这个指定 LOG 级别与当前终端 LOG 等级 console\_loglevel 来决定是不是向终端打印。如果调用者未将日志级别提供给 printk,那么系统就会使用默认值 KERN\_WARNING "<4>". 由于默认值存在变化,所以在使用时最好指定 LOG 级别。我们可以选择性地输出 LOG,比如平时我们只需要打印 KERN\_WARNING 级别以上的关键性 LOG,但是在调试的时候,我们可以选择打印 KERN\_DEBUG 等以上的详细 LOG。而这些都只需要通过命令修改 proc 文件系统中默认日志输出级别文件,命令如下。

```
mtj@ubuntu :~$ cat /proc/sys/kernel/printk
4 4 1 7
mtj@ubuntu :~$ cat /proc/sys/kernel/printk_delay
0
mtj@ubuntu :~$ cat /proc/sys/kernel/printk_ratelimit
5
mtj@ubuntu :~$ cat /proc/sys/kernel/printk_ratelimit_burst
10
```

第一项定义了 printk API 当前使用的日志级别。这些日志级别表示了控制台的日志级别、默认消息日志级别、最小控制台日志级别和默认控制台日志级别。printk\_delay 值表示的是 printk 消息之间的延迟毫秒数(用于提高某些场景的可读性)。注意,这里它的值为 0,但它是不可通过/proc 设置的。printk\_ratelimit 定义了消息之间允许的最小时间间隔(当前定义为每 5 秒内的某个内核消息数)。消息数量是由 printk\_ratelimit\_burst 定义的(当前定义为 10)。如果你拥有一个非正式内核,而又使用有带宽限制的控制台设备(如通过串口),那么这非常有用。注意,在内核中,速度限制是由调用者控制的,而不是在 printk 中实现的。如果一个 printk 用户要求进行速度限制,那么该用户就需要调用 printk\_ratelimit 函数。

内核消息都被保存在一个 LOG\_BUF\_LEN 大小的环形队列中。关于 LOG\_BUF\_LEN 的定义如下。

```
CONFIG_LOG_BUF_SHIFT=18
```

以下是记录缓冲区的操作。

- 当消息被读出到用户空间时，此消息就会从环形队列中被删除。
- 当消息缓冲区满时，如果再有 `printk()` 调用，新消息将覆盖队列中的旧消息。
- 在读写环形队列时，同步问题很容易得到解决。

这个纪录缓冲区之所以称为环形，是因为它的读和写都是按照环形队列的方式进行操作的。

## 2. oops 与 ksymoops、kallsyms

应用程序或内核线程崩溃都会产生 oops 消息，发生 oops 消息时，系统不会死机，而是在终端或日志中打印 oops 信息。当使用 NULL 指针或不正确的指针值时，通常会引发一个 oops 消息，这是因为当引用一个非法指针时，页面映射机制无法将虚拟地址映像到物理地址，处理器就会向操作系统发出一个“页面失效”的信号。内核无法“换页”到并不存在的地址上，系统就会产生一个“oops”。oops 消息显示发生错误时处理器的状态，包括 CPU 寄存器的内容、页描述符表的位置，以及其他的一些辅助信息。

在 Linux 中，调试系统崩溃的传统方法是分析在发生崩溃时发送到系统控制台的 oops 消息。可以将消息发送到 ksymoops 实用程序中，它会试图将代码转换为指令，并将堆栈值映射到对应的内核符号。如回溯线索中的地址，会通过 ksymoops 转化成名称可见的函数名。还可以直接在内核编译时添加 kallsyms，就不需要使用 ksymoops 工具了。

## 3. klogd

klogd 提供了许多信息来帮助分析，为了使 klogd 正确工作，必须在 /boot 中提供符号表文件 System.map。如果符号表与当前内核不匹配，klogd 就会拒绝解析符号。内核本身的打印信息会通过 /proc/kmsg 显示，或者调用 dmsg 命令查看。

## 4. BUG\_ON

在代码里面总能看到 `BUG_ON()`、`WARN_ON()` 这样的宏，类似我们日常编程里面的断言 (assert)。其在 `include/asm-generic/bug.h` 里定义。

## 5. 动态调试

动态调试是指通过动态地开启或禁止某些内核代码来获取额外的内核信息。

首先内核选项 `CONFIG_DYNAMIC_DEBUG` 应该被设置。所有通过 `pr_debug()`/`dev_debug()` 打印的信息都可以动态地显示或不显示。

可以通过简单的查询语句来筛选需要显示的信息，源文件名、函数名、行号（包括指定范围的行号）、模块名、格式化字符串，将要打印信息的格式写入 `<debugfs>/dynamic_debug/control` 中。

## 6. 插入探测点：kprobe

内核中的对应机制是 `kprobe`。`kprobe` 是由 IBM 的 `Dprobe` 项目发展而来的。使用过 `iptables` 的用户都知道，命令中定义的规则实际上是在正常的内核代码执行流程中插入的钩子函数。而钩子函数不但能用来过滤和执行变化，还能用来调试代码逻辑。

`kprobe` 支持 3 种调试方式，第一种叫作 `kprobe`，用来在特定的内核代码位置添加代码（相当于自己手动添加了内核代码再编译执行）；第二种是 `jprobe`，可以用于调试内核函数的传入参数；第三种是 `kretprobe`，用于调试内核函数的返回值。

`kprobe` 是通过回调的形式执行的，因此就有 3 种可能的回调方式：执行前、执行后和出错时的回调。

使用 `kprobe` 一方面要在内核配置中打开 `kprobe` 选项，另一方面要在用户端使用工具，这个工具是 `systemtab`。使用这个工具编写的脚本在执行时，其会在后台编译生成内核模块，插入内核与 `kprobe` 的内核部分协作完成功能。

那么 `kprobe` 是如何做到的呢？利用异常。当定义了插入点后，`kprobe` 的内核部分就会在内存中将插入点附近的代码保存起来，用触发异常的代码替换，当执行到这里的时候异常被触发，回调函数被执行，执行完毕后恢复被保存的原代码继续执行。

## 7. 断点执行

### (1) kgdb

`kgdb` 提供了一种使用用户端的 `gdb` 程序调试 Linux 内核的机制。使用 `kgdb` 可以像调试普通的应用程序那样，在内核中进行设置断点、检查变量值、单步跟踪程序运行等操作。使用 `KGDB` 调试时需要两台机器，一台作为开发机（`Development Machine`），另一台作为目标机（`Target Machine`）。两台机器之间通过串口或者以太网网口相连。串口连接线是一根 RS-232 接口的电缆，在其内部两端的第 2 脚（`TXD`）



与第 3 脚 (RXD) 交叉相连, 第 7 脚 (接地脚) 直接相连。在调试过程中, 被调试的内核运行在目标机上, gdb 调试器运行在开发机上。

kgdb 补丁的主要作用是在 Linux 内核中添加了一个调试 Stub。调试 Stub 是 Linux 内核中的一小段代码, 提供了运行 gdb 的开发机和所调试内核之间的一个媒介。gdb 和调试 stub 之间通过 gdb 串行协议进行通信。gdb 串行协议是一种基于消息的 ASCII 码协议, 它包含了各种调试命令。当设置断点时, kgdb 负责在设置断点的指令前增加一条 trap 指令, 当执行到断点时控制权就转移到调试 stub 中去。此时, 调试 stub 的任务就是使用远程串行通信协议将当前环境传送给 gdb, 然后从 gdb 处接受命令。gdb 命令告诉 stub 下一步该做什么, 当 stub 收到继续执行的命令时, 将恢复程序的运行环境, 把对 CPU 的控制权重新交还给内核。

这个流程与 kprobe 很相似, 整个过程也与 IDA Pro 程序的 andriod\_server 程序运行方式类似。

## (2) kdb 内核调试器

kdb (Kernel Debug) 是 SGI 公司开发的遵循 GPL 的内建 Linux 内核调试工具。标准的 Linux 内核不包括 kdb, 需要从网上下载对应标准版本内核的 kdb 补丁, 对标准内核打补丁, 然后编译打过补丁的内核代码。目前 kdb 支持包括 x86 (IA32)、IA64 和 MIPS 在内的体系结构。

kdb 调试器是 Linux 内核的一部分, 它提供了检查内存和数据结构的方法。通过附加命令, 它可以格式化显示给定地址或 ID 的基本系统数据结构。kdb 当前的命令集可以完全控制内核的操作, 包括单步运行一个处理器、在指定的指令执行处理暂停、在访问或修改指定虚拟内存的位置暂停、在输入/输出地址空间对一个寄存器访问处暂停、通过进程 ID 跟踪任务、指令反汇编等。

## 8. 其他方法

### (1) kexec

kexec 是一个工具集, 允许用户从当前正执行的内核装载另一个内核。用户可以使用 shell 命令 “yum install kexec-tools” 安装 kexec 工具包 (或者是 apt-get 等其他安装方式), 安装后就可以使用 kexec 命令了。

kexec 通过系统调用使用户从当前内核装载并启动进入另一个内核。在当前内核中, kexec 执行 BootLoader 的功能, 在 kexec 启动期间, 硬件层面的初始化不会被执行 (例如 BIOS)。

内核编译配置需要配置“CONFIG\_KEXEC=y”来支持 kexec。首先用 kexec 将调试的内核装载进内存，然后用 kexec 启动装载的内核。

```
kexec -lkernel-image --append=command-line-options
--initrd=initrd-image
```

在上述命令中，参数 kernel-image 为装载内核的映射文件。该命令只支持非压缩的内核映射文件 vmlinuz；参数 initrd-image 为启动时使用 initrd 映射文件；参数 command-line-options 为命令行选项，应来自当前内核的命令行选项，可从文件“/proc/cmdline”中提取。该文件的内容如下。

```
root@ubuntu:~# cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.2.0-93-generic
root=UUID=e5a0ec8c-80b6-4a08-8944-d63cee7c936c ro consoleblank=0 vga=788
processor.max_cstate=1 intel_idle.max_cstate=1
crashkernel=384M-2G:128M,2G-:256M
```

内核装载后，用 kexec 的 -e 选项启动装载的内核，并在新的内核中运行。

## (2) kdump

kdump 是基于 kexec 的崩溃转储机制 (kexec-based Crash Dumping)，当内核需要转储时，如系统崩溃时，kdump 使用 kexec 捕捉内核。kexec 启动一个新的内核（捕获内核），这个内核的目的就是捕获之前运行内核（生产内核）的运行信息，主要是内存内容。

## (3) SysRq 魔术组合键打印内核信息

SysRq 魔术组合键是一组按键，由键盘上的“Alt+SysRq+[CommandKey]”这 3 个键组成，其中 CommandKey 为可选的按键。SysRq 魔术组合键根据组合键的不同，可提供控制内核或打印内核信息的功能。默认 SysRq 组合键是关闭的，可用下面的命令打开此功能。

```
# echo 1> /proc/sys/kernel/sysrq
```

关闭此功能的命令如下。

```
# echo 0> /proc/sys/kernel/sysrq
```

## (4) 命令 strace

命令 strace 显示程序调用的所有系统调用。用户使用 strace 工具，可以看到这

些调用过程及其使用的参数，了解它们与操作系统之间的底层交互。当系统调用失败时，一些错误信息可以被更清楚地展示。

strace 的另一个用处是解决和动态库相关的问题。当对一个可执行文件运行 ldd 时，它会告诉你程序使用的动态库和找到动态库的位置。

### (5) 内核锁验证器

内核锁验证器 (Kernel lockvalidator) 可以在死锁发生前检测到死锁，即使是很少发生的死锁。它将每个自旋锁与一个键值相关，相似的锁仅处理一次。加锁时，查看所有已获取的锁，并确信在其他上下文中没有已获取的锁，在新获取锁之后被获取。解锁时，确信正被解开的锁在已获取锁的顶部。

当加锁动态发生时，锁验证器映射所有加锁规则，该检测由内核的 spinlocks、rwlocks、mutexes 和 rwsems 等锁机制触发。当内核锁验证器检测到一个新加锁场景，它检查新规则是否违反正存在的规则集，如果新规则与正存在的规则集一致，则加入新规则，内核正常运行。如果新规则可能创建一个死锁场景，那么这种创建死锁的条件会被打印出来。

### (6) 硬件模拟

vmware 或者 virtualbox 创建的虚拟机，可以模拟两台电脑的互连。skyyeye 也可以模拟硬件，它是一个纯粹的硬件模拟平台，对于开发 arm 的嵌入式系统上的 Linux 十分有用。使用 UML 调试 Linux 内核是在本机的 Linux 上调试 Linux 的好方法，User-ModeLinux (UML) 可以在用户端作为进程运行 Linux 内核，如此可以轻松地使用 gdb 等用户端调试程序。

## 7.7 PAM和Apparmor

### 1. PAM

PAM 是 Linux 的基础和基本安全组件，是一个应用程序，这个应用程序给其他的应用程序提供安全服务。其他应用程序希望使用 PAM 提供的安全服务需要在它的代码中加入 PAM 的 API 接口，所以 PAM 本质上是一个提供给其他进程使用的库 (libpam.so)。

使用 PAM 库会使用 /etc/pam.d/ 目录下的配置文件，这些配置文件里面就是规定这个应用是如何使用 PAM 的。只要 /etc/pam.d/ 目录存在，libpam.so 库就会忽略



/etc/pam.conf 配置文件，如图 7-5 所示。

```
root@ubuntu:/etc/pam.d# ls
atd  chpasswd common-account common-password common-session-noninteractive login  other polkit-1 quagga su
chfn chsh common-auth common-session cron  newusers passwd .ppp sshd sudo
root@ubuntu:/etc/pam.d#
```

图 7-5

由于所有的应用程序都使用同一个库，而这个库的配置文件的目录是同一个目录，所以在这个目录下的任意文件都通过语法指定是哪个服务。比如在这些配置文件中会有如下所示内容。

```
passwd auth required pam_passwd_auth.so.1
```

就是规定 `passwd` 程序需要经过认证。因此，`libpam.so` 的逻辑是把 `libpam.so` 的配置不区分应用程序进行通用的解析，内部用语法进行区分，并不是每一个应用单独规定自己的配置内容，虽然在实际操作中很多程序会有以自己名字命名的配置文件，比如 `login`。这种设计就使得安全配置可以通过配置文件的修改来直接完成，这也就决定了应用程序在使用 `libpam.so` 库的时候并不会使用任何与实际的安全策略有关的 API，而一定都是通用的。Shane Watts 写过一个示例代码可以很容易地看清楚 PAM 的使用，示例如下。

```
#include <security/pam_appl.h>
#include <security/pam_misc.h>
#include <stdio.h>
static struct pam_conv conv = {
    misc_conv,
    NULL
};
int main(int argc, char *argv[])
{
    pam_handle_t *pamh=NULL;
    int retval;
    const char *user="nobody";

    if(argc == 2) {
        user = argv[1];
    }
    retval = pam_start("check_user", user, &conv, &pamh);
```

```

if (retval == PAM_SUCCESS)
    retval = pam_authenticate(pamh, 0);    /*检查用户名和密码*/
if (retval == PAM_SUCCESS)
    retval = pam_acct_mgmt(pamh, 0);      /* 检查用户是否过期 */

if (retval == PAM_SUCCESS) {
    fprintf(stdout, "Authenticated\n");
} else {
    fprintf(stdout, "Not Authenticated\n");
}
if (pam_end(pamh,retval) != PAM_SUCCESS) {    /* 关闭 Linux-PAM */
    pamh = NULL;
    fprintf(stderr, "check_user: failed to release authenticator\n");
    exit(1);
}
return ( retval == PAM_SUCCESS ? 0:1 );
}

```

编译的方法是 `gcc pamtest.c -o checkuser -lpam -lpam_misc`。在逻辑上还应该添加一个 PAM 文件 `/etc/pam.d/checkuser`，代码如下。

```

auth        required    pam_unix.so
account     required    pam_unix.so

```

但是现在的 Ubuntu 系统会默认添加一个 `/etc/pam.d/other` 文件，这个文件就是没有指定配置情况的默认认证文件，其中包括这里用到的基础密码认证。所以这里也可以不创建这个文件，如图 7-6 所示。

```

root@ubuntu:~# ./checkuser archerbroler
Password:
Authenticated
root@ubuntu:~# ./checkuser archerbroler
Password:
Not Authenticated
root@ubuntu:~#

```

图 7-6

有一个经典的入侵手法是使用一个软链接就能让 `sshd` 无密码登录，代码如下。

```
ln -sf /usr/sbin/sshd /tmp/su; /tmp/su -oPort=6666;
```



这就是利用了 `sshd` 和 `su` 程序在 PAM 认证中的认证流程的不一样，当然各个系统是不同的，这取决于 PAM 和两个服务本身的配置。这个案例从另一方面说明了 PAM 系统使用启动文件的名称作为认证的入口是很不安全的。

## 2. Apparmor

我们知道原始的 Linux 安全体系是以用户为单位控制的，包括设置每个文件的访问权限标志位也是以用户和组来区别的。Apparmor 的思路是以应用进行安全限制，而不是以用户来进行安全限制。以下是笔者机器上的 `tcpdump` 的 Apparmor 配置，即使不理解细节语法，也能很容易地看懂这个配置文件的意义。典型的工作就是控制进程的文件权限和网络权限，这里还进行了 `audit` 的权限设置，代码如下。

```
root@ubuntu:~# cat /etc/apparmor.d/usr.sbin.tcpdump
# vim:syntax=apparmor
# Last Modified: Wed Feb  3 07:58:30 2009
# Author: Jamie Strandboge <jamie@canonical.com>
#include <tunables/global>

/usr/sbin/tcpdump {
    #include <abstractions/base>
    #include <abstractions/nameservice>
    #include <abstractions/user-tmp>

    capability net_raw,
    capability setuid,
    capability setgid,
    capability dac_override,
    network raw,
    network packet,

    # for -D
    capability sys_module,
    @{PROC}/bus/usb/ r,
    @{PROC}/bus/usb/** r,

    # for finding an interface
    @{PROC}/[0-9]*/net/dev r,
```





```
/sys/bus/usb/devices/ r,  
/sys/class/net/ r,  
/sys/devices/**/net/* r,  
  
# for -j  
capability net_admin,  
  
# for tracing USB bus, which libpcap supports  
/dev/usbmon* r,  
/dev/bus/usb/ r,  
/dev/bus/usb/** r,  
  
# for init_ethernarray(), with -e  
/etc/ethers r,  
  
# for USB probing (see libpcap-1.1.x/pcap-usb-linux.c:probe_devices())  
/dev/bus/usb/**/[0-9]* w,  
  
# for -z  
/bin/gzip ixr,  
/bin/bzip2 ixr,  
  
# for -F and -w  
audit deny @{HOME}/.* mrwkl,  
audit deny @{HOME}/.* / rw,  
audit deny @{HOME}/.*/** mrwkl,  
audit deny @{HOME}/bin/ rw,  
audit deny @{HOME}/bin/** mrwkl,  
owner @{HOME}/ r,  
owner @{HOME}/** rw,  
  
# for -r, -F and -w  
/**. [pP][cC][aA][pP] rw,  
  
# for convenience with -r (ie, read pcap files from other sources)  
/var/log/snort/*log* r,
```



```
/usr/sbin/tcpdump mr,  
  
# Site-specific additions and overrides. See local/README for details.  
#include <local/usr.sbin.tcpdump>  
}
```

我们使用 `apparmor_status` 命令能看到当前 Apparmor 系统的保护状态，这不同于 PAM，Apparmor 并不需要用户的代码级别进行集成接口，它保护的是系统资源，而 PAM 保护的是业务安全。这也就决定了 Apparmor 就像 Android 系统中常见的权限系统：一个应用的执行需要各种权限，这个权限需要程序外部的系统资源，但是在内部不确定的位置会使用到。因此有了 Apparmor，即使 `tcpdump` 程序被替换为其他的恶意木马应用，也不用担心它会影响到系统，除非它有能力将 Apparmor 的配置文件修改掉。不过一般系统具备了 `tcpdump` 这种文件的修改能力，也就拥有了 `root` 权限，这些防御也就都不可靠了。因此，这种形式的安全在工业界的实际应用中也有限的，人们关注更多的是如何防止外部进入和提权。

## 7.8 内核安全

内核安全最常见的是 `rootkit` 修改系统调用，将自己作为一个模块隐藏在内核中。检测方法也很简单，通过对比实际运行的内核的 API 调用表和 `sysmap` 文件（或者是 `/proc/kallsyms`），得到内核是否被感染的结论。`rootkit` 后来还发展了技术，对系统调用的修改改成了用到的时候临时修改。一个很清晰的示例是 `wukong` 这个 `rootkit`，代码见 `github`：<https://github.com/hanj4096/wukong>。

如果使用覆盖同地址的方法，检查的时候就需要实际检查内存的前几个字节。还可以使用直接替换调用表的方法，自己在内核中生成一份自己的调用表，然后 `hook` 系统调用的入口，这样符号表没有任何改变。这种检测方法就是检测 `system_call` 的调用是否实际使用调用表的地址。

## 7.9 常用安全工具和项目

在 Linux 系统中有一些常用的安全工具，这里列举的工具大部分都是开源的安



全工具，付费的不在大部分研究者的考虑范围内。

### 1. 验证系统

- PAM: 几乎在所有的发行版上都是默认安装的。

### 2. 内核安全

- Grsecurity: 专注于内核的安全，给内核提供了很多安全补丁。
- ExecShield: 设置不可执行的标志位 (现代的 x64 硬件携带的就可以不用软件了, x86 是不带的)。
- Linux Intrusion Detection System (LIDS): 使用 MAC (Mandatory Access Control) 做的内核安全补丁集。

### 3. 网络防御

- TCP wrappers: 可以通过它来设置哪些远端的 ip 允许访问, 哪些不允许访问。
- Iptables (还有流量控制的 tc): 综合性的网络安全工具 (Vuumuur、Turtle Firewall、Firestarter、FireHOL、Shorewall 可以作为配置接口), 还可以直接使用 shelter 等基于 iptables 的防火墙工具, 拥有众多的模块。
- Arpon: 专门防 arp 欺骗 (arpwatch 可以用来记录 arp 日志)。
- Clearos: 防火墙 (内容过滤、入侵检测、带宽控制等, 提供常见的服务器), 这是一个单独的操作系统, 并不是软件。类似的还有 IPCop、openwall (用的比较多), Network Security Toolkit (NST) 专门用作网络监控的 live cd。
- Bro: 静态监听并且分析网络数据流量, 发现入侵行为。

### 4. 综合性系统安全分析

- Tiger: 功能比较弱, 全部是使用 shell 写的, 只是检查一些文件和配置。
- Lynis: 可以用来取代 tiger, 功能一般, 但是能输出一些对于系统整体的审计报告。
- Nessus (OpenVAS 是比较权威的综合性主机扫描工具): Nessus 后来闭源, 开源的版本由 OpenVAS 继续。
- OSSIM: 集合和各种开源软件。
- ACARM-ng: hids 系统。
- SELinux、Apparmor: 用户端综合防御系统。





## 5. 密码破解

- John: 结合/etc/passwd 和/etc/shadow, 直接用彩虹表爆破系统密码。
- Hydra: 远端服务密码的破解。
- RainbowCrack: 提前计算常用密码的彩虹表, 然后直接查找破解。

## 6. 病毒查杀

- Clamav: 扫描本机上的病毒。
- p3scan: 扫描 E-mail 中的病毒。

## 7. 文件完整性检查

- Tripwire: 最常用的文件完整性检查工具, 有变动通知。
- AIDE: 目标是取代 Tripwire。
- OSSEC: 不是专门做文件完整性检查的工具, 而是一个通用工具。其他的不太常用的类似的工具还有 AFICK、Aide。

## 8. Rootkit 和 web shell 检测

- shell-detector: Python 脚本, 专门检测机器中的 web shell。
- Chkrootkit: 检测机器中的 rootkit。
- OSSEC: 检测机器中的 rootkit (还有其他功能)。
- Rkhunter: 检测 rootkit、后门、恶意软件 (rootkit 和 chkrootkit 可以两个一起用)。

## 9. 系统监控

- Nagios: 强大的主机资源监控报警系统。
- Ntop (iptraf、iftop、tcpdump 等): 网络流量监控, 下一代监控是 ntopng。

## 10. 服务分析

- Denyhosts: 分析 SSH 的日志来发现有没有人试图入侵和已经入侵成功。

## 11. 数据加密

- CipherShed: 磁盘加密工具。
- Peazip: 文件压缩和加密工具。
- Steghide: 信息隐写工具。



- Stunnel: 在 ssl 中隐藏 TCP 连接。
- TrueCrypt: 磁盘加密工具。

## 12. 辅助工具

### (1) 高级网络工具

- Dsniff: 可以嗅探插入数据包, 用于获得用户机的口令和密码。
- Ettercap: 局域网嗅探与入侵工具, 以太网攻击“神器”。
- Nping: 全方位的 ping (包含了 arping、fping、ping、hping 的全部功能和其他扩展功能), 是 nmap 程序包的一部分。
- Socat/ncat: nc 的升级版, 功能更强大, 但是不如 nc 使用便捷。ncat 是 nmap 程序包的。
- Ngrep/tcpdump: 两者各有所长, 使用 ngrep 进行业务调试更方便; tcpdump 适合更复杂的分析。
- Iptraf: 看当前的网络数据流。
- SSH: 远程登录, 端口转发。
- Scapy: 网络探测“神器”。号称可以取代 hping 和部分的 nmap、arp spoof、arp-sk、arping、tcpdump、tshark、p0f。与 ethcap 的功能有一定的重合。
- Rsync: 文件一致性保持本分工具。
- Tc/Tcng: 流量控制“神器”。Tcng 是一个编译器, 专门用于流量控制领域, 但是在 Ubuntu 16 中已经没有了这个软件包。
- Ethtool: 查看详细的网卡信息。
- Nemesis: 伪造数据包, 可以用来生成攻击数据包。
- Mtr: ping 和 traceroute 的结合体。
- Snort: 侦测网络通信协议, 发现威胁。

### (2) 漏洞和网络扫描工具

- Angry IP Scanner: 快速扫描中的“神器”(masscan 也算得上“神器”), 用来全网扫描。
- Nmap: 最通用的端口扫描工具, 扫描能力很强。
- Hping: 另外一个常用的扫描和探测网络的“神器”。
- Sara: 扫描网络发现漏洞。
- Nikto: Web 服务器漏洞和弱密码扫描。



- W3af: Web 漏洞扫描。

### (3) 商业扫描工具

- Core Impact: 比较厉害的漏洞扫描工具。
- Nexpose: 著名的开源漏洞扫描工具 msf 的商业版本。
- GFI LanGuard: 不少用户从 nexpose 迁移过来。
- Saint: 曾经的开源“神器”，现在闭源。扫描网络发现漏洞。
- Canvas: 漏洞扫描与利用。
- MedusaL: 暴力破解各种常见服务程序。

### (4) 渗透测试操作系统

- Backbox: 基于 Ubuntu 的发行版。
- Kali: 基于 debian。工具比 Backbox 多，而且有 bug boundy。
- Pentoo: 另外一个渗透测试系统。





# 8

## 第 8 章

### 网络

#### 8.1 网络架构

Linux 网络部分有非常多的内容，充分体现了其兼容并包的特点。大部分的知识我们平时都用不到，这里给出一个功能清单式的概览：802.1d 以太网桥、802.1Q/802.1ad VLAN 支持、IP 负载压缩、ANSI/IEEE 802.2 LLC type 2 支持、MPLS 支持、DANH (Doubly attached node implementing HSR)、Switch 交换功能支持、Open Vswitch 软交换支持、CCITT X.25 包层支持、LAPB 数据链路支持、用来取代 UDP 协议的 DCCP 支持、用来取代 TCP 的 SCTP 支持、RDS (在 Infiniband、iWARP 或 TCP 网络上提供可靠的、顺序的服务传输协议)、TIPC (在高可信度网络中 TCP 的耗费太大，爱立信提出的新的可靠数据传输服务，与 SCTP 定位类似)、ATM、IP over ATM、LAN 模拟、RFC1483/2684 桥支持。

DECnet 是一种协议族。与 IP 对应的网络层协议是 DRP (DECnet Routing Protocol)，与 TCP 对应的传输层是 NSP (Network Service Protocol)。与 TCP/IP 族不同的是，DECnet 协议族在传输层之上还有会话层 SCP (Session Control Protocol) 和网络管理层 MOP (Maintenance Operation Protocol)。DECnet 早期与 TCP/IP 竞争，但最终被 TCP/IP 协议族完全击败。IPX/SPX 是另一种协议族。IPX 与 IP 的功能类似，SPX 与 TCP 的功能类似，但没有与 UDP 类似的协议。目前仍有很多用户在用



这个协议族，因为在 LAN 环境中，传输层 SPX 的效率比 TCP 高。这套协议最早是 Novell 在 Dos 上实现的，Windows 将 smb/Netbios 构建在 IPX/SPX 之上。另外，NCP 也是广泛使用的网络文件与打印共享协议，不过近期无论是 NetBios 还是 NCP 都已经支持 TCP/IP 协议族了。所以 IPX/SPX 的可取代性变得越来越高，应用就越来越少。还有其他的更加少见的协议族，比如，Phone Network Protocol (PhoNet)、6LoWPAN、AppleTalk 和一些底层的协议族。

网络功能却远远不止上述这些，还有例如 RxRPC、远程过程调用 RxRPC、用于 CPU 之间通信的 CAIF、Netlink 等。

我们通常见到的网络是以太网和无线网络，但是 Linux 是一个大而全的操作系统，其支持很多无线网络，主要有业余无线电、CAN 网络、红外线 (IrDA)、蓝牙 (Bluetooth)、Wi-Fi、WiMAX、RF 开关、Plan 9、NFC 等。以太网的很多东西在其他网络中也是通用的，我们主要讨论常见的以太网。内核编译时会给出很多网络部分的选项，通过设置这些选项可以打开和关闭某些网络功能，但并不意味着 Linux 支持全部的网络功能。

一般是分层讨论计算机网络的。从 OSI 将计算机网络分层后，如果不使用分层的方法，网络将无法阐述。分层让逻辑清楚，易于理解，更重要的是易于实现和工业化（每个公司或软件都可以集中注意力在自己所要实现的某个层次的某个功能上）。

常见的有线物理层，即使用的传输介质有网线、电视线（同轴电缆）和光纤。每一种物理传输线都有其特性，例如可用的频段、吞吐量、传输损耗等，根据不同的情况要选用不同的调制解调方式和划定不同的传输信道。像对网络进行分层一样，所有的传输介质都对可以在线缆中使用的频段进行了划分，这些被划分的频段叫作信道。因此信道管理也是每个传输介质所要对应的逻辑部分。例如信道动态地添加删除、分配、冲突避免等逻辑上的操作构成了 MAC 层。所以可以见到推广某一物理传输介质的组织，一般都会同时定义 MAC 层，有的甚至定义数据链路层。

我们见到的很多网络协议族都是没有链路层的，例如以太网 MAC 层之上直接就是 IP 层，而有的却有链路层，例如 Wi-Fi。这是什么原因呢？这就涉及链路层存在的意义了。数据链路层是在信道之上建立的逻辑连接，这个连接不同于 TCP 层的连接，TCP 层的连接目的是高速高效地传输数据，保证数据可正常到达及其稳定性，而数据链路层的连接通常是用来控制可访问性、计费、认证等链路监视和控制功能的。所以一个开放的网络不需要链路层，因为访问链路是自由的，不受别人控制。

而如果使用 ISP 的服务，或者使用无线提供者的服务，你是否可以接入网络（IP 数据包能否发送出去）取决于服务提供商是否允许你访问网络，这时数据链路层就是有意义的。

而即使接入了网络，在众多节点中，你的数据包如何到达你想要它到达的地方呢？这就需要寻址。就像地图里的各个地点，如果要旅行首先需要道路连接，然后还得知道如何选择道路才能正确地到达目的地。因此，这里面有 3 个要素，即各个节点（地点）的命名（标志）、节点之间的连接（道路），以及如何选择线路。网络中各个节点的质量不一样，节点之间的链路也不一样。网络有分布式的或者集中式的，为了容易控制，Internet 网络路由选择了分层的集中式，而有的局部网络可能就是分布式的。IP 地址的网络地址部分（子网掩码规定的）代表了链路，后面低位的部分代表了节点。网络地址相同的，一般链路上是相连的，不同网络之间通过一个或多个节点互相联系，这个节点一般同时有两个网段的 IP。如此便解决了前两个要素。解决第三个要素是通过路由表，每个节点维护了路由表，记录了什么目的地址的数据包该转发给谁。得到这些路由信息的方法就是通过社会工程（实际组网的分配设置）和一些动态的路由协议（如 RIP、OSPF）等交换通信得到的。

有了网络层完成寻址，数据包就可以到达对方的 PC 了，但是每个 PC 上并不是只有一个程序需要使用网络数据。不同程序的数据必须区分开，这就诞生了链路层的需求。链路层又为每个 IP 地址（每个 PC）定义了端口的概念，一个应用程序使用一个（或多个）端口，在数据包中写入了端口信息就可以被指定的程序所理解和处理了。

程序不只是用网络来发送无意义的数，发送和接收的内容是经过定制的，可以被程序所理解。这就是应用层协议，在链路层之上。

内核网络功能的 core 部分最外层是 3 个接口文件，即 `socket`、`compat`、`sysctl_net`。`socket` 文件定义了操作系统暴露给用户程序的接口；`compat` 是兼容性考虑的特殊 `socket` 接口（主要服务于 `sparc`）；`sysctl_net` 向内核的 `sysctl` 接口注册服务（并没有具体的实现节点，具体的实现在各个内部模块）。

目录中最重要的是 `core` 文件夹，其内部是 `net` 整个部分的基础架构，其他文件下面大多数都是具体的某一种网络协议。这些网络协议有的是我们所知道的因特网上的，例如 `IPv4`、`sctp`；有的是专用网上的，例如 `x25`、`can` 等；有的是无线网络上的，例如 `wimax`、`wireless`、`irda` 等；有的是硬件与硬件之间的局部通信，例如 `caif`；



还有的甚至只是一个文件系统的抽象网络，例如 ceph。所以 net 目录下的网络一般是抽象层次比较高的概念，是指一切对外暴露 socket 接口的功能模块。目录的组织更多的是一种技术上的安排，而不是产品上的。另外，很多内容都是定义在对应的.h 文件中的。

最核心的 core 部分的主要工作就是实现 socket 机制所需要的基础设置，包括以下 3 项。

- 用来放网络数据的 skbuf; 在内核不完全启动时的收发数据框架 netpoll; 过滤数据包的 filter; 对 ethtool 的支持 (就是用户端 ethtool 命令对应的内核代码); 缓存目标地址的 dst\_entry 机制; 监测丢包的 drop\_monitor; 与网络设备 (net\_device) 相关的交互; 一些通用的网络相关的概念 (tso、timestamp、stream); 测试调试相关的接口 (pkgen); 邻居通用管理 (毕竟任何一种网络都有邻居的概念);
- 通用的 socket 部分 (针对各个不同的协议 sock 结构体会派生) 和一些针对 sock 操作的封装实现 (请求分配);
- 对内核各个功能模块的支持工作 (Netlink、sysctl、sysfs、trace、procfs、cgroup、内核通知链、名称空间)。

在 core 的这些实现里，大部分都是辅助性的，核心就是 sock 结构体和 skbuff。

值得注意的是，多数人首先在用户端使用 Linux，在看到内核代码的时候就已经有了很多使用 Linux 发行版的经验。你或许精通 socket 编程，但是对于内核来说，就要忘记 socket 了，因为内核里面根本不存在 socket，也可以叫作 sock。socket 是内核暴露给用户端的编程接口，在内核的网络协议栈里，socket 会被转化为 sock。

由于 TCP/IP 网络的应用实在太广了，所以导致即使在具体协议无关的 core 里面也存在很多 TCP、UDP 和 IP 的影子，但是不要因为这些文件组织就怀疑网络多样化的事实。内核的代码组织更多的是技术性的，而不是产品性的。在 core 里有 stream 和 datagram 两种传输层概念的定义，虽然并不是所有的网络都有传输层。core 看起来更多的是为 TCP/IP 这个强者服务的。这也是为什么在创造 socket 的时候既得指明 SOCK\_STREAM，又得指明 IPPROTO\_TCP。按照函数的解释类型是 SOCK\_STREAM，具体协议是 IPPROTO\_TCP。但是这对用户来说，谁都知道 IPPROTO\_TCP 的类型是 SOCK\_STREAM，需要在这么重量级的参数上指定吗？这就是接口和底层的矛盾所在。接口要假装自己可以适应多种情况，底层也确实有多

种情况，而常出现的基本只有一种情况。那么为其他不常出现的情况留出一个参数是必要的吗？这让很多初学 socket 编程的人感到莫名其妙。同时这也是历史原因和兼容并包导致的问题。

skbuff 用来存放网络中的数据包，这些数据包可以是任意的网络协议，所以 skbuff 也必须支持。这个数据包会从 socket 的最上层一直“下”到硬件设备。因此，其又必须考虑要能同时被各个层次的功能操作。不同的层次对数据包的处理会有各种不同的需求，例如发送了一个 IP 包并不直接删除这个包，因为系统担心万一发送不成功，还得重新构造一遍，不如保留，这就涉及怎么保留的问题。用户向 skbuff 中写数据（通常调用的是 write 或 send），可以分段写入，这又要求 skbuff 有灵活的扩展性。

像这种结构体，如果在 C++这种面向对象的编程思想里要实现上述需求，一定会采用类似 decorator 的设计模式。但在 C 语言里，各个部分的需求功能的支持数据都只能列在一个结构体里，实现一个看起来像 component 的设计模式。下面列出对 skbuff 有使用要求的内核模块。

- netfilter 要识别 skbuff 并且对 skbuff 进行过滤（会在 skbuff 中添加辅助这个功能的加速域）；
- 网络协议的各个层次要对数据进行修改、计算校验、添加头部等操作；
- DMA 要使用 skbuff 中的数据；
- GSO、TSO、LRO、GRO 等硬件加速功能要能识别和使用 skbuff；
- 克隆需求。例如 TCP 不知道是否发送成功，会事先克隆一份（为什么不重用？因为 skbuff 向下走的时候会被修改，既然要克隆，就得决定哪些东西要克隆，哪些东西可以共享）；
- 对 skbuff 打时间戳的需求；
- 与 sock 和 dev 的关联。任何一个 skbuff 必定是来自某个用户的 socket（对应一个内核的 sock），去往或者从某个 net\_device 到达；
- 出于资源管理考虑的组织（要做成链表或 rb\_tree）。

我们可能以为要满足这些需求，那就自己去使用 skbuff 就好了，与 skbuff 结构体有什么关系？这就是 Linux 内核的做事方式（除非你能想出效率更高的方法），所有这些看似外部的需求几乎都会在 skbuff 的结构体中添加域。好像每个实现特定部分的人都以在 skbuff 中添加了自己的域为荣一样，最后导致 skbuff 结构体正如你看到的那样巨大无比。

但是，我们知道 `skbuff` 作为数据的载体，最大的功能就是组织数据。其结构体中记录了各个头部在数据中的位置，是否被克隆过、是否计算过校验等状态和指示信息（当然，这些信息都是分别在上百万内核代码的某处被更新和维护的），但最关键的是，其数据是以可灵活调整大小的 `fragment` 组织的（类似 `scatter list`）。其定义的大部分函数也都是为了方便地修改结构体中的域，你当然可以不用函数而直接通过修改结构体的域来完成几乎全部的工作，但是不推荐直接修改数据结构的方式，因为数据结构太庞杂。

所以在笔者看来，`skbuff` 被“虐待”成什么样不要紧，重要的是“虐待”它的功能模块有哪些。它只是用来记录这些模块战果的“记事本”和“战场”而已。

## 8.2 socket

### 8.2.1 socket 简介

`skbuff` 可以跟踪数据包的整个生命周期，`sock` 则是跟踪一个 `socket` 的整个生命周期。什么是 `socket`？它是内核中的一个资源实体，利用这个实体就可以访问网络了。也就是说，网络协议栈对外呈现的并不是一个面向过程的函数调用，而在概念上反而是一个由类生成的一个个 `socket` 对象，通过这一个个对象，用户可以使用调用对象的方法访问网络。

为什么需要这个对象呢？因为每个应用程序对网络使用的都是个事务，这个事务可能包含很多个不同的步骤，也可能是长期的。而任何一个函数调用都只能完成整个事务的一部分。所以每个事务就需要变量来存储当前的状态。对于网络调用来说，就是监听的是哪个端口；这个端口是否允许重用；这个事务的所有者是哪个或哪些进程，绑定的是哪个设备，使用的是什么协议族；源地址和与这个事务有关的缓存、内存、资源（例如可以同时监听的连接数 `backlog`）、超时时间等。这些参数对每个函数调用来说都只是变量，但是对于单个事务来说，就是这个事务存续期间的不可变量。这就是 `sock` 结构体存在的意义。综上所述，`sock` 结构体代表的是网络事务。

在 Linux 中，协议栈看起来只有一个，但是随着虚拟化的流行，在 Linux 中也



支持多个协议栈但没必要把协议代码拷贝多份，其秉承的还是面向对象的思想。`struct net` 这个结构体本身就代表了一个协议栈，生成一个对象就是一个协议栈。说白了就是由一个协议栈变成多个协议栈，也就是定义一个变量集。换句话说，一个协议栈本身也描述了一个事务，从这个协议栈的出生到死亡。同时存在多个协议栈也就不足为奇了。

多个 `net` 结构体存在的最大用途在于虚拟化应用场景，这是一个命名空间概念在网络协议栈的体现（其他的还是文件系统空间、进程 `PID` 空间）。但如果在别的地方看到网络的命名空间不要被专业术语给迷惑了，其实就是协议栈对象。

网络通信服务一般由操作系统提供，使用这个服务的一般是进程。我们知道在正常情况下应用层是建立在传输层之上的，也就是说程序使用的协议一般是传输层协议。传输层常见的有 `TCP` 和 `UDP`，分别代表了保证传输质量的和不保证传输质量的两种类型。其他的还有 `SCTP`（希望取代 `TCP`）、`DCCP`（希望取代 `UDP`）等传输协议。所以进程编程就是要选择合适的传输协议，然后调用传输协议来进行数据通信。

像每一层都给上一层提供统一的稳定接口一样，传输层也给应用层提供了统一的调用接口，即 `socket`。不过更进一步说，这个 `socket` 的概念后来被广泛拓展，现在几乎变成了使用所有网络相关服务的接口了。就如同 `TCP/IP` 被用在了 `ATM` 网络上一样，好的技术规范是会被广泛采纳的。

## 8.2.2 类型与接口

`socket` 有很多种类型，分别工作在不同的网络协议层，但是 `socket` 对外的函数和数据接口是基本一致的。常见的 `socket` 类型有用于直接发送 IP 数据包的 `Packet Socket`；有用户本机进程通信的 `UNIX Domain Socket`；有用于 `IPSec` 上通信的 `PF_KEYv2 socket`；有用于在 `TCP/UDP` 上通信的 `socket`；有用于虚拟机与主机通信的 `Virtual Socket`；还有用于用户与内核通信的 `Netlink`。不同类型的 `socket` 的区别在于调用的网络服务不一样，但是操作接口都是一样的。

```
int socket(int family, int type, int protocol)
```

`socket` 究竟是指什么呢？所有操作系统为了有效地管理资源，并且能被用户程序有效地索引，都会为资源用数字编号命名，这叫作句柄。我们自己写用户进程时

可以使用指针，但是内核与用户空间之间是不能用指针引用的。通过一个句柄的分配和查询，让无论是打开的文件还是生成的 socket 都可以有唯一的身份 ID。socket 是一个句柄，对于内核来说，用户必须提供这个句柄才能使用 socket 相关的功能（如发送数据），所以生成 socket 句柄是所有用户进行 socket 编程的第一步。这个函数如下所示。

```
#include<sys/socket.h>
int socket(int family, int type, int protocol)
```

由于 socket 相关的调用是通用的，但是用户总得指定传输协议，指定的方法就在于这个函数。family 指定协议族，比如 TCP/IP 协议族、OSI 协议族或者 AppleTalk 协议族。协议族的不同，决定了内核使用的整个协议栈的不同。type 参数的设置是因为所有的传输层协议无非只有两种：数据流式和数据包式。虽然本质上所有的数据都是通过数据包传输的，但是在传输层看来，如果数据包可以有序且不遗漏地到达，那么就是数据流了。因此数据流式的协议（例如 TCP）都会提供额外的传输控制，而数据包式的协议（例如 UDP）一般只是起到了不同端口定位不同程序的功能。因此，常见的 type 参数有 SOCK\_STREAM 和 SOCK\_DGRAM 两种取值。又由于 socket 的推广，所以其还支持 SOCK\_SEQPACKET（SCTP）、SOCK\_RAW（IP）等类型。protocol 就是指明具体的传输协议。其实完全可以只用一个参数来编号所有的传输协议，但是为了容易区分，还是分开成两个参数。

由于 UDP 没有连接概念，而 TCP 有连接概念，两者又要拥有同样的对外接口函数，那么如何设计接口呢？目前的方法是按照 TCP 的需求设计接口，UDP 只需要使用其中的一部分，然而即使调用了例如 connect 这种面向连接概念的接口，UDP 也能正常处理 connect 函数调用，甚至可以完成某些功能。

socket 是内核中的一个拥有状态的对象，刚创建出来的时候默认是 CLOSED 状态的，TCP 在 server 调用了 listen 函数之后，socket 会变为 Listen 状态，之后还会随着 TCP 连接的建立和关闭而切换为不同的状态。可以看出 socket 虽然同时为 TCP 和 UDP 服务，但是主要是考虑了 TCP 的需求，UDP 是 socket 提供服务的子集。

网络进程分为 server 和 client 两端。server 负责监听连接，client 负责发起连接（特殊情况不考虑）。所以对于这两端，无论是 UDP 还是 TCP，需要的函数调用接口是不一样的。但是双方都需要先选择自己的 IP 地址和端口（因为无论是 client 还是 server 都可能有多 IP 地址），这个函数是 bind。然而现在很多 socket 实现起来都

可以做到智能选择，所以不调用 `bind` 也是可以的，但是如果不成功可能就得自己手动选择了。但是对于 `server` 来说，随机选择的端口客户端无法知道连接哪个端口，所以 `server` 还是得调用 `bind`。

```
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addlen);
```

第一个参数 `sockfd` 是 `bind` 的套接字；第二个参数 `myaddr` 是地址和端口的结构体；第三个参数 `addlen` 指明长度。其实不用指明长度内核也知道长度（因为结构体就是它定义的）。端口和 IP 地址可以都指定，也可以都不指定（设为 0），或者可以指定一个。如果不指定，内核随机选取了端口号，选择的这个端口号也不会被函数返回，必须要调用 `getsockname()` 手动获得。而不指定 IP 地址（指定为 `INADDR_ANY`），内核会等到 TCP 建立连接或者 UDP 发出数据所使用的 IP 地址来设定为 `socket` 之后使用的 IP 地址，但是主监听 `socket` 还会继续使用通配地址。这个函数有以下 4 种调用场景，如表 8-1 所示。

表 8-1

网 络 端	调用场景
UDP server	只监听指定的 IP：端口。未指定的随机选取
UDP client	只从指定的 IP：端口发送数据。未指定的随机选取
TCP Server	只监听指定的 IP：端口。未指定的随机选取
TCP client	只从指定的 IP：端口发送数据。未指定的随机选取

## 1. 服务器端

```
int listen(int sockfd, int backlog)
```

这是唯一一个只可以由 TCP `server` 端调用的函数。因为这个函数的主要功能是识别三次握手。

我们要分清的一点是 `socket` 是内核资源，`listen` 操作也是由内核完成的，内核完成三次握手完全不需要用户程序的参与。调用了 `listen` 就相当于告诉内核，为我监听网络中的 TCP 连接，完成了三次握手再叫我。之后进程就可以陷入睡眠了。

内核既然要完成三次握手，那么就要考虑多个用户同时连接的情况。事实上还有 SYNC Flood 攻击的情况。三次握手的设计导致内核在收到 SYNC 后回复 SYNC/ACK，并且要等待客户端的继续回复。由于网络环境的不可靠，用户不一定会回复，回复的时间也不一定。所以内核为监听三次握手的 `socket` 维护了两个队列，一个是正在



等待 client 最后回复的队列；另一个是已经成功完成三次握手的队列。backlog 参数没有固定的意义，甚至各个 Linux 发行版的意义都不同，但都是用来计算这两个队列的大小的参数。在 Linux 下一般用 proc 文件系统的 /proc/sys/net/core/ somaxconn 文件控制 TCP 连接的最大连接数目。

已经成功建立的队列中如果有了新的条目，内核就会唤醒用户进程将已经建立的 socket 给进程。如果正在等待 client 回复的队列满了，内核将无法再继续接收新的 TCP SYNC 连接请求，此时 server 将不回复（如果回复 RST，用户端就会放弃。不回复则会重试，说不定一会就有了）。通常出现这种情况，要么是设置的 backlog 不够大，要么是受到了 TCP Flood 的攻击。

```
int accept(int sockfd, struct sockaddr*cliaddr, socklen_t *addrlen)
```

当客户端调用 connect 成功（三次握手成功），server 端的服务器进程就可以得到对应的 socket。但是 socket 资源属于内核管理，内核可以将服务进程唤醒，但是没有办法主动地把已经建立好的 socket 交给用户进程，需要用户进程主动发起，这就是 accept 系统调用。

实际上，内核完全可以主动把已经连接的 socket 放到用户进程预先制定的用户内存中，然后再唤醒用户进程。但是如此设计，用户进程调用 listen 醒来就需要检查醒来的原因，还需要去指定的缓存获得已经建立的 socket，这块缓存一次缓存多少个合适呢？这一切都增加了设计上的复杂度。最终采用的设计是内核只负责唤醒在 listen 的进程，进程被唤醒后调用 accept 函数会去向内核尝试获得一个已经完成三次握手的 socket。这样所有的流程安排都掌握在进程手中。还是那个原则，提供机制而不提供策略。

这里需要注意的是，accept 得到的返回值是建立了三次握手的 socket，但是不是其在监听 TCP 连接的 socket 呢？对比一下就会发现两者的句柄是不相同的。listen 的 socket 在用户没有进入 listen 时暂停，新产生的 socket 表示了一个已经建立的 TCP 连接。通过该连接，用户与 server 可以互相通信。由于一般情况下 server 不止服务于一个用户，所以 accept 得到的 socket 一般会重新建立一个线程或者进程与用户进行通信。原 server 进程继续调用 listen 等待新的用户连接。

## 2. 客户端

```
int connect(int sockfd, const struct sockaddr* servaddr, socklen_t
addrlen);
```

`listen` 是服务端等待三次握手的过程，`connect` 则是客户端发起三次握手的过程。对于 UDP 客户端，`connect` 则是选择了之后发送数据包的 IP 地址和端口，没有实际的网络操作。

三次握手首先发送 SYNC，对于客户端来说，就要等待回复的 SYNC/ACK，因此这个等待与服务器的情况类似，很有可能其也收不到回复。在发送第一个 SYNC 之后可能收到目的地址不可达的 ICMP，或者收到 RST 回复，也可能什么都收不到。对于不同的回复不同的操作系统会有不同的反应。例如什么都收不到可以有超时重传，重传时规定重传次数。

### 8.2.3 Linux socket 连接模型

socket 连接模型大体分为三类：阻塞、多进程、I/O 多路复用。最基本的就是阻塞模型，进程阻塞 `listen`，有了连接之后，进程自己调用 `accept` 函数，自己接收并处理数据。由于其在处理数据的过程中不能继续 `listen`，所以一次只能服务一个用户。在改进的模型中，当 `accept` 到一个新的连接时，就生成了一个新的线程或进程，用新的线程或进程去处理这个连接，而自己继续 `listen`。

由于 `fork` (`vfork`) 的系统资源成本比较高，所以改进可以预先生成一个线程池。里面有很多待服务的进程。这时可以设计为让主进程自己 `listen`，也可以让各个线程一起 `listen`，谁先从内核取到数据谁服务，其他继续 `listen`。如果内核的一个 socket 有了连接，会一次叫醒所有正在 `listen` 的进程（惊群效应），如此开销也不小。所以一般会采用主线程统一 `listen` 的方法。

上述的 3 种模型一次都只可以 `listen` 一个 socket，然而一个进程有时候需要同时 `listen` 多个 socket，此时需要 `select` (`pselect0`) 和 `epoll` (`poll`)。

`select`、`pselect`、`poll`、`epoll` 都是 IO 多路复用。这里的 IO 不但包括网络通信，还包括与本机磁盘的通信（读写）。因此，使用这几个函数监听的描述符可能是 socket 在等待连接，也有可能是写入磁盘的操作等待完成。

`pselect` 只是 `select` 的 POSIX 函数接口版本，内容上没有太大差别。`poll` 也和 `select`

流程功能一样，一般网络应用都直接使用 `select`。但是 `select` 系列调用有致命的缺点，就是每次调用都需要把要监听的句柄集拷贝到内核中，内核还需要完整地遍历所有句柄来查看哪个有新的动态。这样在要监听的 `socket` 有很多时就会有问题。由于 `select` 在处理完成后还会继续调用，又触发一次拷贝和遍历，导致连接太频繁，开销更大。而且 `select` 支持的句柄只有 128 个。

`epoll` 是专门用来改善 `select` 的弊端的。针对每次调用都要传递句柄集的情况，`epoll` 定义了 `epoll_create` 函数，在调用等待之前把句柄都一次性拷贝到内核中，之后要修改就调用 `epoll_ctl`。这样每次调用等待连接的函数 `epoll_wait` 时就不需要频繁拷贝，并且 `epoll` 不是遍历查看句柄的状态，而是注册回调函数。当句柄状态发生变化时，就会调用对应的回调函数。由轮训到中断模型的变化无疑可以提高执行效率。

## 8.3 IP

TCP/IP 是目前通用性最高的网络系统。早期各个硬件体系大多数都会定义自己的通信协议，并且以太网上也曾经存在很多其他种类的通信协议（例如 IPX），但是随着产业的发展，所有的网络系统都在逐渐接受 TCP/IP 协议族。Linux 内核对这部分的支持也已经很完整了。

### 8.3.1 IP 管理

管理机器的 IP 地址看起来是一件非常简单的事情，但是当你更加深入学习时就会发现其实这并不简单。举个例子，某台机器上已经有 IP 地址了，但是我再添加一个 IP 地址（`ip address add 192.168.99.37/24 dev ens33`）。用该命令查看网卡的 IP 地址，如图 8-1 所示。

```

2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:a1:1f:6c brd ff:ff:ff:ff:ff:ff
    inet 192.168.128.129/24 brd 192.168.128.255 scope global ens33
        valid_lft forever preferred_lft forever
    inet 192.168.99.37/24 scope global ens33
        valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:fe01:1f6c/64 scope link
        valid_lft forever preferred_lft forever

```

图 8-1



可以看到执行完这条命令后，ens33 口有两个同时存在的 IP 地址。然而我们用 ifconfig 命令只能看到一个 IP 地址，如图 8-2 所示。

```
root@ubuntu:~/sdb1/yunweishi/build/pragram/infocollector# ifconfig
ens33:  Link encap:Ethernet  HWaddr 00:0c:29:e1:1f:6c
        inet addr:192.168.128.129  Bcast:192.168.128.255  Mask:255.255.255.0
        inet6 addr: fe80::20c:29ff:fe01:1f6c/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:4771647 errors:0 dropped:0 overruns:0 frame:0
        TX packets:5317566 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:1996321614 (1.9 GB)  TX bytes:2691656822 (2.6 GB)

lo:      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:65536  Metric:1
        RX packets:550 errors:0 dropped:0 overruns:0 frame:0
        TX packets:550 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1
        RX bytes:59676 (59.6 KB)  TX bytes:59676 (59.6 KB)
```

图 8-2

此时对这个新添加的 IP 地址执行 ping 命令是生效的，如图 8-3 所示。

```
root@ubuntu:~/sdb1/yunweishi/build/pragram/infocollector# ping 192.168.99.37
PING 192.168.99.37 (192.168.99.37) 56(84) bytes of data:
64 bytes from 192.168.99.37: icmp_seq=1 ttl=64 time=0.123 ms
64 bytes from 192.168.99.37: icmp_seq=2 ttl=64 time=0.030 ms
^C
--- 192.168.99.37 ping statistics ---
```

图 8-3

然后编写程序，使用 Netlink 接口从内核中或得到的接口列表中发现有两个相同名字的接口，每个接口分别有一个 IP 地址，虽然与 ip 命令的结果是一致的，但是组织方式明显不同。也就是说这两个 IP 地址在内核看来属于两个接口，但在 ip 命令看来属于一个接口，在 ifconfig 命令看来却只存在一个。我们用 ifconfig 命令操作时就只能生成一个类似 ens33.0 的虚拟接口，然后给这个接口赋值。

说到出现这种情况的原因就要谈谈 Linux 的发展过程了。ifconfig 是早期 Linux 内核的组织方式对应的工具，后来随着内核的进步，出现了新的组织方式和命令 ip，而人们由于惯性还没有完全切换到新的使用思路上来。

内核启动的时候可以提供参数指定 IP 地址，也可以在启动时通过 bootp 或 DHCP 动态获得 IP 地址信息。这对于无盘操作系统是十分重要的。BOOTP 是 DHCP 的早期版本，功能比较简单，现在多见于嵌入式系统。DHCP 的流程比较复杂，协议定义了一系列消息进行网络通信。静态地址在 RFC 中被称为外部配置。其没有经过网

络发现步骤，直接配置了 IP 地址，如果出现冲突或者不在一个域，电脑将不能联网。还有一种情况就是之前通过 DHCP 获得过 IP 地址，在本机启动的时候客户端就会使用 DHCPINFORM 来直接通知服务器自己希望使用的 IP 地址，服务器回复 DHCPACK 消息来完成其他参数的配置，静态缓存使得 DHCP 过程与平常的 DHCP 过程不一样。如果静态地址发送 DHCPINFORM 的时候，DHCP 的 server 发现这个 IP 地址已经被其他人使用了，就会重新进行 DHCP 流程。还有一种是不需要配置的，但是也不属于静态的地址配置方式，叫作 zeroconf，这种方式在苹果系统中多见（Bonjour），在 Linux 系统下也会使用（Avahi）。现代 IPv6 的出现直接使用了 zeroconf 的思路，启动时直接使用 MAC 地址构造 IPv6 的地址，虽然也有 DHCPv6，但是 IPv6 完全支持不需要地址配置的情况。

### 8.3.2 IP 隧道

#### 1. 原理

隧道是指自己的数据包完全架构在另一个协议的头部之上。IP 隧道利用 IP 协议的路由和寻址能力，在 IP 层以上封装自己的完整协议包。例如 IP 层之上可能是 MAC 层，然后是 IP 层，也有可能为了加密将上层的数据协议加密的密文，或者是 ATM 层。

内核中常用的隧道有 IPIP（内核模块是 ipip.ko）、GRE（generic routing encapsulation，内核模块是 ip\_gre.ko）两个在 IP 层上进行封装的，以及 PPTP、L2TP 两个在数据链路层进行封装的。GRE 是通用路由封装协议，可以在网络层（IP 或 IPX）之上封装数据包，是 VPN 的第三层隧道协议。GRE 利用管道，是管道之上的一种通信协议。使用 GRE 进行通信的双方必须按照 GRE 协议的要求封装包和解封装包。确切地说，GRE 是 IP 隧道之上的一种应用，但不限于 IP 隧道。

#### 2. IPIP、GRE

最初的封装概念很简单，即为了在 TCP/IP 网络中传输其他协议的数据包。设想 IPX 协议或 X.25 封装的数据包如何通过 Internet 网进行传输？在已经使用多年的桥接技术中通过在源协议数据包上再套上一个 IP 协议头来实现，形成的 IP 数据包通

过 Internet 后卸去 IP 头，还原成源协议数据包，传送给目的站点。对源协议数据来说，就如同被 IP 带着过了一条隧道。利用 IP 隧道来传送的协议包也包括 IP 数据包，本文主要分析的 IPIP 封包就是如此，从字面上来理解 IPIP，就是把一个 IP 数据包又套在一个 IP 数据包里。为什么要这么做呢？好像是多此一举。其实不然，见过一些应用就会明白了，移动 IP（Mobile-IP）和 IP 多点广播（IP-Multicast）是两个常见的例子。目前，IP 隧道技术在构筑虚拟专网（Virtual Private Network）中也显示出极大的魅力。本文也对利用 IP 隧道技术构筑 VPN 做了简单的设想。

但是 Linux 支持的隧道不止 IPIP 一种，IPIP 这种隧道只能在 IP 之上封装 IP 协议。另外一种常用的协议是 GRE，其上层可以封装任何协议。如果封装的上层协议是 ip，那头部就是 ip:gre:ip，而 IPIP 的头部则是 ip:ip。另外，其他种类的封装协议，如 mpls、ipsec 在服务器领域使用较少。

还有一个很新的 tunnel 技术就是 vxlan，从内核 3.12 版本开始支持。它是把传统的路由器 vlan 进行了扩展，用在广域网上组管道，不再是在 IP 层之上的封装，而是在 UDP 层之上从 MAC 层开始封装。也就是说，协议的头部变为了 mac:ip:udp:vxlan:mac:ip:udp。当然，vxlan 上层封装可以是任何的网络类型，甚至可以是跨越不同的 MAC 类型网络。vxlan 与其典型的应用 open vswitch 一起构成了云时代虚拟网络的基础。其不但可以桥接网络，并且可以设置跨广域网的 vlan，在跨广域网的虚拟局域网中实现 QoS，还可以进行丰富的流量监控和数据包分析（其封装之上是完整的 mac:ip:udp 头部），这些对于以前的 tunnel 盲传是不可想象的。

Internet 的研究者多年前就感到需要在网络中建立隧道，最初的理解是在网络中建立一条固定的路径，以绕过一些可能失效的网关。可以说隧道就是一条特定的路径。这样的隧道是通过 IP 报头中的源路由选项来实现的，在目前看来，这个方法的缺陷十分明显。要设置源路由选项就必须知道数据包要经过的确切路径，而且目前多数路由器在工程实现中都不支持源路由。

另一个实现隧道的机制是开发一种新的 IP 选项，用来表明源数据包的信息，源 IP 头可能成为此选项的一部分。这种隧道的意义与我们所说的隧道已十分接近。但它的不足之处在于要对目前 IP 选项的实现和处理做较大的修改，同时也缺乏灵活性。

最后一种常用的实现方法是开发一种新的 IP 封包协议，仍然套用当前的 IP 头格式。通过 IP 封包，无须指明网络路径，封包就能透明地到达目的地。也可以通过封包空间把未直接连接的机器绑在一起，从而创建虚拟网络。这种方法易行、可靠、



可扩展性强，Linux 采用了这一方法，也是目前我们所理解的隧道思想。

封包协议的实现原理十分简单。先看看通过隧道传送的数据包在网络中是如何流动的，为了叙述简便，下面把在隧道中传送的 IP 数据包称为封包。两个端点设备分别处于隧道的两端，分别起到打包（封装）和解包（解封）的作用，在整个数据包的传送路径中，除了隧道两端的设备，其他网关把数据包看成一个普通的 IP 包进行转发。端点设备就是一个封包基于的两个实现部件——封装部件和解封部件。封装部件和解封部件（设备）都应当同时属于两个子网。封装部件对接收到的数据包加上封包头，然后以解封部件地址作为目的地址转发出去；而解封部件则在收到封包后，还原原数据包，转发到目的子网。

### 3. PPP、PPPoE、PPTP、L2TP

PPP 是一个数据链路层协议，能控制上层通信的使能、加密和计费。例如未经过 PPP 认证的网络，IP 数据包是发送不了的。PPP 改进自 SLIP，是数据链路层协议，其并不关心网络层是 IP 协议还是 IPX，或是 Appletalk 协议。

PPP 的过程分为链接建立和网络两部分。链接建立可以认证，网络部分是认证成功后协商如何加密传输上层的数据。

PPP 协议是点对点的，换句话说，PPP 协议没有寻址功能。但是在实际的网络环境中，没有寻址功能的协议基本没用。所以人们在 PPP 的外面封了一层以太网头部，供最开始的时候节点向 ISP 建立 PPP 连接寻址用。这种网络叫作 PPPoE。

以太网的寻址无法经过路由器，虽然有了以太网编址，让 ISP 可以一个服务端服务多个客户端，但是无法经过路由器的缺点使得 ISP 必须把认证服务器在物理上与用户放得很近。因此就诞生了新的需求，是否可以将 PPP 认证服务器放到网络的任何地方呢？这就必然用到网络层在因特网上寻址的能力和传输层的传输能力。PPTP 就是如此，其认证过程使用 TCP（在 PPP 连接建立前，前端路由器只放行路由到认证服务器的数据包），其网络过程使用 IP 上的 GRE 封装。

随着 PPTP 的使用，人们发现其与 TCP/IP 协议族绑定的缺点是无法在其他网络中使用。所以 L2TP 诞生了，其不依赖于任何的网络层和传输层，只要网络层能够路由数据包，传输则由 L2TP 协议本身完成。还有一个改进就是 PPTP 一条链路只能有一个会话，但是 L2TP 可以有多个，因此一台 PC 使用多个 L2TP 应用就成为了可能（例如多个窗口同时远程登录）。PPP 状态迁移图，如图 8-4 所示。

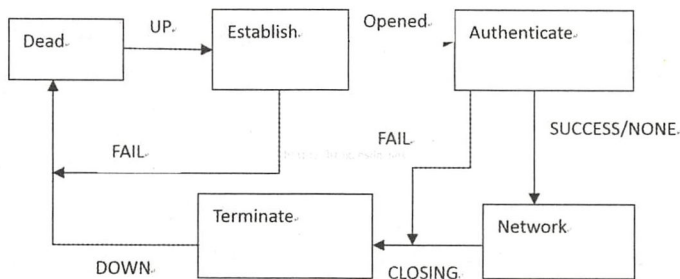


图 8-4

- **Dead:** 用于表示物理层状态。当物理层准备好（例如载波检测或者管理员配置）时，将从此阶段切换到 Establish 阶段，并向 LCP 自动机发送一个 UP 信号。当连接结束时（例如将猫拔出），将回到本阶段。这个阶段是用来表示物理层的连通性的。
- **Establish:** 在本阶段，LCP 使用网络参数协商并建立连接。当在 Network 和 Authenticate 阶段收到 LCP Configure-Request 时将回到 Establish 阶段。如果接下来需要经过认证阶段，则本阶段必须协商认证的方式。
- **Authenticate:** 建立连接后，立即进入认证阶段。认证阶段只允许特定的数据包通过，当认证通过时将进入 Network 阶段，失败则进入 Terminate 阶段。本阶段为可选阶段。
- **Network:** 完成上述步骤后，网络中的每一层都调用 NCP 协议进行每层的网络配置。当 NCP 进入 open 状态后，所有的网络包就可以在已经建立的链路上通行了。
- **Terminate:** 本阶段只接受 LCP 数据包，以 LCP 发送终止数据包开始。本阶段由 LCP 沟通关闭连接，并进入 Dead 阶段。

#### 4. 认证方式

PAP 为基本的两次握手认证，口令以明文的方式传输。拨号用户发送用户名和密码到接入服务器，接入服务器通过 RADIUS 协议到 RADIUS 服务器上，查看是否有此用户，以及口令是否正确，然后发送相应的响应。

CHAP 为三次握手认证，口令为密文传输，在认证过程和网络通信过程会周期性地发送认证。CHAP 拨号用户发送用户名到接入服务器中，接入服务器发送随机产生的报文交给拨号用户，拨号用户用自己的口令，用 MD5（可选）算法进行运算，

传回密文，接入服务器从 RADIUS 服务器取得的用户口令和随机报文，用 MD5（可选）算法运算，比较二者的密文，根据比较的结果返回失败或成功的响应。

## 5. XFRM 和 IPSec

在网络通信中，通信双方的认证和通信内容的加密是很重要的需求。这个可以在上层解决，也可以在相对较下的网络层次解决。IPSec 就是在 IP 层之上，传输层之下的解决方案。

其主要解决的内容是通信双方的认证、通信内容的加密、加密密钥的交换、通信数据的压缩。

在通信数据的压缩上，其采用的协议名是 PayloadCompression Protocol (IPComp)。其本质上就是通信的双方将 IP 层以上的数据进行压缩和解压缩。认证和加密都是通过压缩算法的应用实现的（密码学基础），关键是密钥的交换。这部分是 IPSec 协议中唯一不在内核中实现的部分，叫作 IKE (Internet Key Exchange)。

IPSec 由于要加密通信，其实就相当于在 IP 层维护链接，相当于 IP 层之上的数据链路层。链接需要针对每条链接进行记录，以便后续的数据包经过合适的处理。在内核中存储与链接相关的信息依靠两个表，即 Security Policy Database (SPD) 和 Security AssociationDatabase (SAD)。如果你使用命令行在用户端运行 IPSec，那么操作的主要对象就是这两个表。这两个表是用户影响 IPSec 发挥作用的唯一机制，会接收和发送 Netlink 的 NETLINK\_XFRM 消息，并且按照要求进行更新。用户端可以使用 Netlink 接口给 XFRM 模块发送信息。已经写好的比较通用的用户端进程可以操作 XFRM 的是 `ip xfrm` 命令。

## 6. Linux 路由表，路由策略，路由查找

在内核中路由表有两种，一个是缓存路由 (FIB)，自动学习生成自动管理，用户没必要去干预，但是内核还是提供了方法让用户可以去清空它。用户不能设置它的项，但是可以根据这个缓存更新的原理从外部影响它；另外一个路由表，一共有 256 个子表，在内核中是一个数组，可以通过配置让内核使用其中的一个或者多个路由表。默认的是使用 0、254 号、255 号这三个路由表。一般大家关心的都是 254 号的主路由，`route` 命令看到的和操作的都是这个路由表。255 号是 local 路由，还包含了广播地址等。0 是全路由，还包括了 IPv6 的信息，是最全面的。但是 254 号的主路由是最容易看到的，也是用户最关心的，所以 `route` 命令只操作这个路由表。选择使用哪个路由表的操作叫作留有策略，这个也是可以通过 `ip rule` 命令配置的。



如图 8-5 所示的命令显示了不同的路由表信息，我们平时操作的路由表的路由条目都是默认 254 号的路由表，如图 8-6 所示。

```

root@ubuntu:~# ip route list table 0
default via 172.26.80.254 dev ens33
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.26.80.0/24 dev ens33 proto kernel scope link src 172.26.80.23
broadcast 127.0.0.0 dev lo proto kernel scope link src 127.0.0.1
local 127.0.0.0/8 dev lo table local proto kernel scope host src 127.0.0.1
local 127.0.0.1 dev lo table local proto kernel scope host src 127.0.0.1
broadcast 127.255.255.255 dev lo table local proto kernel scope link src 127.0.0.1
broadcast 172.17.0.0 dev docker0 table local proto kernel scope link src 172.17.0.1 linkdown
local 172.17.0.1 dev docker0 table local proto kernel scope host src 172.17.0.1
broadcast 172.17.255.255 dev docker0 table local proto kernel scope link src 172.17.0.1 linkdown
broadcast 172.26.80.0 dev ens33 table local proto kernel scope link src 172.26.80.23
local 172.26.80.23 dev ens33 table local proto kernel scope host src 172.26.80.23
broadcast 172.26.80.255 dev ens33 table local proto kernel scope link src 172.26.80.23
fe80::64 dev ens33 proto kernel metric 256 pref medium
unreachable default dev lo table unspec proto kernel metric 4294967295 error -101 pref medium
local ::1 dev lo table local proto none metric 0 pref medium
local fe80::20c:29ff:fe01:1f6c dev lo table local proto none metric 0 pref medium
ff00::8 dev ens33 table local metric 256 pref medium
unreachable default dev lo table unspec proto kernel metric 4294967295 error -101 pref medium
root@ubuntu:~# ip route list table 254
default via 172.26.80.254 dev ens33
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.26.80.0/24 dev ens33 proto kernel scope link src 172.26.80.23
root@ubuntu:~# ip route list table 2
root@ubuntu:~# ip route list table 255
broadcast 127.0.0.0 dev lo proto kernel scope link src 127.0.0.1
local 127.0.0.0/8 dev lo proto kernel scope host src 127.0.0.1
local 127.0.0.1 dev lo proto kernel scope host src 127.0.0.1
broadcast 127.255.255.255 dev lo proto kernel scope link src 127.0.0.1
broadcast 172.17.0.0 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
local 172.17.0.1 dev docker0 proto kernel scope host src 172.17.0.1
broadcast 172.17.255.255 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
broadcast 172.26.80.0 dev ens33 proto kernel scope link src 172.26.80.23
local 172.26.80.23 dev ens33 proto kernel scope host src 172.26.80.23
broadcast 172.26.80.255 dev ens33 proto kernel scope link src 172.26.80.23
root@ubuntu:~#

```

图 8-5

```

root@ubuntu:~# ip route show
default via 172.26.80.254 dev ens33
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.26.80.0/24 dev ens33 proto kernel scope link src 172.26.80.23
root@ubuntu:~#

```

图 8-6

每一个路由表里面的路由类型都可以有 6 种，最常见的是单播和网段类型的，具体如下。

- 单播：目的地址是某一个 IP 地址，一般是手动添加的。
- 网段：这个路由类型是最常见的，表示到达某个网段需要从哪里发送出去。
- nat：nat 也是路由的一种，它会修改掉 IP 的地址域为要到达的地址。nat 也是一种形式的路由，这种路由和 iptable 的 nat 是同时存在的两种不同的机制。
- unreachable：网络不可达类型的路由。我们经常看到不可达，通常是因为没

有配置到目的地址，或者是配置的不对。但是还可以单独配置一个不可达类型的路由，即使它是可达的。

- **prohibit**: 禁止类型的路由。到某个地址的路由默认都是添加的如何到达，但是也可以添加如何禁止。同样是到某个网段或地址的路由，可以在某个网口上设置禁止某个网段，这个与实际的到不了不同，是查路由的时候由路由表告知这个网段是被禁止的。
- **blackhole**: 到达目标网段的所有数据包都可以查到，但是都会直接被丢弃。即这是一个欺骗的路由条目。你以为你查到了，并发出去了，其实数据包都被悄悄丢掉了。

如图 8-7 所示，添加了一个不可达类型的路由。这些种类的路由由于功能和路由策略重合，同时使用比较混乱，所以如果要想实现复杂的路由，就应该使用路由策略规则，而不是这里的不可达类型的路由。

```

root@ubuntu:~# ip route add unreachable 123.125.114.144
root@ubuntu:~# ip route list
default via 172.26.80.254 dev ens33
unreachable 123.125.114.144
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.26.80.0/24 dev ens33 proto kernel scope link src 172.26.80.23
root@ubuntu:~# ping 123.125.114.144
connect: No route to host
root@ubuntu:~# ip route del 123.125.114.144
root@ubuntu:~# ip route list
default via 172.26.80.254 dev ens33
169.254.0.0/16 dev ens33 scope link metric 1000
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1 linkdown
172.26.80.0/24 dev ens33 proto kernel scope link src 172.26.80.23
root@ubuntu:~# ping 123.125.114.144
PING 123.125.114.144 (123.125.114.144) 56(84) bytes of data:
64 bytes from 123.125.114.144: icmp_seq=1 ttl=49 time=37.3 ms
64 bytes from 123.125.114.144: icmp_seq=2 ttl=49 time=37.4 ms
^C
--- 123.125.114.144 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 37.348/37.397/37.447/0.199 ms
root@ubuntu:~#

```

图 8-7

路由表的查询匹配算法一般是 LPM (longest prefix match)，这种算法适合于不同精细度的网段，允许匹配最精细的网段设置，如果没有更精细的网段则匹配当前的。最不精细的就是 0.0.0.0 网段，可以匹配全部的网段。

每一个路由表都对应一个路由策略，默认的路由策略最简单，就是查询表。查询表默认有 3 个路由策略，如图 8-8 所示。



```

root@ubuntu:/home/liujingyang# ip rule show
0:      from all lookup local
32766:  from all lookup main
32767:  from all lookup default
root@ubuntu:/home/liujingyang#

```

图 8-8

所以，添加一个除了 0、254 号、255 号之外的路由表之后，该路由表不会正常工作，路由表只是数据库，是否查询和怎么查询都是由路由策略决定的。自己添加了路由表之后要想让这个路由表被查询，需要添加一个对应的路由策略。默认的路由策略都是 lookup，就是我们通常所说的查询行为，还有其他的路由策略行为，如下。

- **nat**: 查询到的路由是用来做 nat 的。对应的路由表中一般有很多 nat 类型的路由表。
- **unreachable**: 所有在对应的路由表中查到的路由条目都给出 unreachable 的答案。
- **prohibit**: 所有在对应的路由表中查到的路由条目都给出 prohibit 的答案。
- **blackhole**: 所有在对应的路由表中查到的路由条目都直接被丢弃。

路由策略从第一个开始向后查询，进入查询每个策略对应的路由表，如果查到了就采取对应的路由策略规定的行为。

如图 8-9 和图 8-10 所示，可以看到不同版本的 ip 命令在路由策略上的变化，显然新的版本去掉了 reject、prohibit 等操作，realm 可以跟 tc 配合一起用于流量控制，大部分人常用的仍是默认的 table 查表方法。

通过使用路由策略，使得对不同的网络查找不同的路由表，这里指定的是查找表 2，而我们的表 2 目前是空的。所以这个路由策略就会被跳过，没有命中表 2 中的任何一个条目。因为只要命中一个就会返回 prohibit，从而拒绝这次路由查找。

```

root@ubuntu:/home/liujingyang# ip -V
ip utility, iproute2-ss111117
root@ubuntu:/home/liujingyang# ip rule add help
Usage: ip rule [ list | add | del | flush ] SELECTOR ACTION
SELECTOR := [ not ] [ from PREFIX ] [ to PREFIX ] [ tos TOS ] [ fwmark FWMARK[/MASK] ]
           [ iif STRING ] [ oif STRING ] [ pref NUMBER ]
ACTION := [ table TABLE_ID ]
           [ prohibit | reject | unreachable ]
           [ realms [SRCREALM/]DSTREALM ]
           [ goto NUMBER ]
TABLE_ID := [ local | main | default | NUMBER ]
root@ubuntu:/home/liujingyang#

```

图 8-9



```

root@ubuntu:~# ip rule add help
Usage: ip rule [ list | add | del | flush | save ] SELECTOR ACTION
       ip rule restore
SELECTOR := [ not ] [ from PREFIX ] [ to PREFIX ] [ tos TOS ] [ fwmark FWMARK[/MASK] ]
           [ iif STRING ] [ oif STRING ] [ pref NUMBER ]
ACTION := [ table TABLE_ID ]
          [ realms [SRCREALM/]DSTREALM ]
          [ goto NUMBER ]
SUPPRESSOR := [ suppress_prefixlength NUMBER ]
              [ suppress_ifgroup DEVGROUP ]
TABLE_ID := [ local | main | default | NUMBER ]
root@ubuntu:~# ip -V
ip utility iproute2-rs151103

```

图 8-10

路由会先在缓存（FIB）中查找，找不到则到路由表中查找。但是这个在路由表中查找时并不单纯地去查表，流程比较复杂。

以前的路由查找，只是单纯地根据目的 IP 地址来进行 LPM 匹配查询，而现在的策略路由支持根据其他的域，比如源地址、tos、端口等来决定匹配的策略（这些不同的判断域叫作 selector）。当然，路由表还是单纯的地址匹配，支持多种匹配的是路由策略（rule）。

## 8.4 TCP

### 8.4.1 TCP 存在的原因

TCP 希望数据按序无损失地传输，只要有 TCP 这个协议的需求，就有其带来的问题。问题是如何保证按序到达和完全到达呢？要保证速度又如何设计机制呢？最终 TCP 的设计者设计的机制是 ARQ，就是同时发送好几个包，对方选择确认。确认的是流，而不是数据包，这倒也符合 TCP 设计的初衷。

为实现 ARQ 的目的，就需要设计一个机制使一端能够应答确认数据流的某个位置，而不是某个包，这个机制就是 sequence 编号。socket 接口本质上是半双工的，也就是说数据通信在发送时就不能接收，接收到了才开始发送，或者是发送后进入接收状态等待接收。既然通信的双方都可以发送数据包，那么双方各有一个 sequence。这个 sequence 表示当前发送的数据包的序号。同时每个 TCP ACK 数据包中还有对对方 sequence 的确认，告诉对方自己目前收到的是哪个位置的数据，这个域就是 ACK。

双方通过把自己的接收情况和发送情况尽可能早地告诉对方，好让对方适时调整发送频率。

在正常情况下机制会很好地工作，可是一旦一方发现对方收到的最新的 sequence 号是自己很早就发送出去的呢？一旦一方发现自己好久没有收到来自对方的包呢？产生第一个问题的原因可能是自己发送到对方的网络阻塞了，产生第二个问题的原因可能是对方没发或者对方发来的网络也是阻塞的，数据包被路由器大量丢掉了。

## 8.4.2 TCP 的连接状态

### 1. 连接的建立与关闭

TCP 是面向连接的，这个面向连接不是为了面向连接而面向的，毕竟面向连接需要额外的成本去维护。连接是数据可靠传输的副作用，要实现数据的可靠传输，通信的两端就需要建立信道，才可以在信道上进行数据控制并且区别于其他信道（这里的信道是逻辑上的通信链路）。

既然连接是无可奈何的产物，那么就涉及如何建立连接、关闭连接和存储连接信息（典型的是连接的列表和状态）。大部分阅读本书的人都应当比较熟习 TCP 的状态图，这里就不给出状态图了。

### 2. TCP 连接建立

TCP 连接就是著名的三次握手。为何要进行三次握手呢？因为连接要让双方都确保建立。TCP 认为确保的方式是双方都既能成功发送又能成功接收数据包。client 发起 TCP 连接（SYNC），等待返回 SYN/ACK。如果接收到返回包，对于 client 来说，其发送的数据包成功，并且也能成功接收。对于 server 来说，由于数据通常都是先由 client 发送的，当其收到第一个 SYN 时，它已经知道自己可以成功地接收数据包了，还要验证能否成功发送，并且让 client 知道自己可以接收，此时其回复 SYN/ACK 数据包。对于 server 来说，此时还不可以确定自己能够成功发送数据包。因此，client 会回复 ACK，当 server 收到后，就确认自己既能发送又能接收了。链路是通的，就可以建立，这就组成了三次握手。

其实三次握手本质上是确保对于双方来说收发链路都是通的，而在大部分情况下理论上两次握手就足够了。因为 client 完全可以确定自己收发都可以，server 可以

确定自己能够接收。而数据又是从 client 首先发送的, 所以 server 完全可以等到收到 client 的数据再确认连接的成功建立, 这样也就省去了第 3 个数据包。Linux 内核中有一个快速 TCP 的子系统, 就是在第三次握手的同时携带数据信息。

还可以通过其他的连接来标记一条可用链路, 从而可以直接对链路添加信任, 就不需要经过握手了。但是也得考虑 IP 可达并不代表端口可达。但是毕竟是有优化空间的。我们知道, TCP 是为广域网设计的, 其基础是对通信链路的不信任。随着网络质量的提高, 这种算法确实有改进的空间。但是由于其优秀的设计, 让其生命力非常顽强。TCP 的设计是选择出来的, 是在实际的环境中形成的最优解, 除非实际的环境发生了巨大的变化, 否则 TCP 的地位不会被撼动。

还有一种建立连接的握手方式是六次握手, 非常少见, 如图 8-11 所示。仅仅发生在 client 和 server 同时向对方的同一个端口发起连接的时候 (RFC1379), 所以一般用于特殊目的。

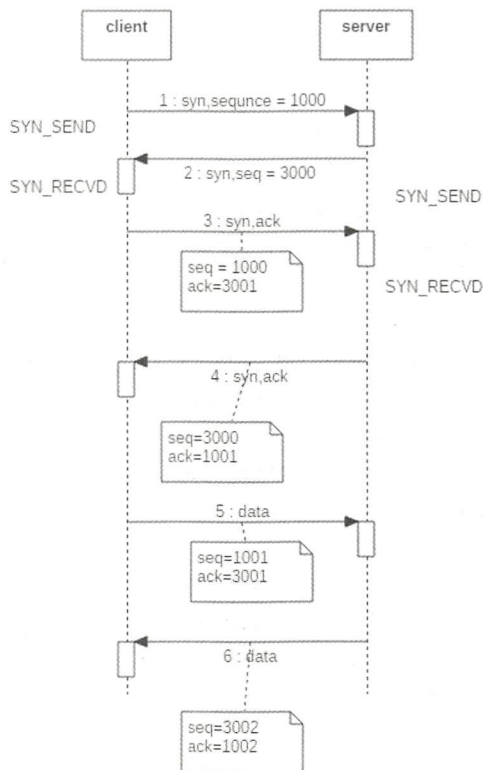


图 8-11



### 3. 六次握手实验

#### (1) 实验内容

Client 172.27.227.84 正常使用 telnet 命令试图与 Server 172.27.229.114 7331 建立连接。Server 上运行了截断内核包处理流程的虚拟服务器。该服务器接收到 SYN 的时候，会立即回复 SYN 并等待回复，在收到 SYN/ACK 之后，回复 SYN/ACK。并且接着回复 ACK。

#### (2) Server 代码

```
print "Waiting for SYN from client ..."  
syn = sniff(iface="eth4", filter="tcp port 7331", count=1)  
ip_packet = IP()  
ip_packet.src = server_ip  
ip_packet.dst = syn[0][IP].src  
tcp_packet = TCP()  
tcp_packet.dport = syn[0][TCP].sport  
tcp_packet.sport = server_port  
tcp_packet.flags = "S"  
tcp_packet.seq = initial_sequence  
t = ip_packet/tcp_packet  
print "Received client SYN, sending server SYN ... "  
rep = srl(t, verbose=False)  
if rep:  
    tcp_packet.flags = "SA"  
    tcp_packet.ack = syn[0][TCP].seq + 1  
    tcp_packet.seq = initial_sequence + 1  
    t = ip_packet/tcp_packet  
    print "Received client SYN+ACK, sending SYN+ACK ..."  
    send(t, verbose=True)  
    tcp_packet.flags = "A"  
    tcp_packet.ack = syn[0][TCP].seq + 1  
    tcp_packet.seq = initial_sequence + 2  
    t = ip_packet/tcp_packet  
    send(t, verbose=True)  
    print "Done."
```

#### (3) 测试的结果与分析

[illegible]

#### 4. 连接释放

双方都为该 TCP 连接分配了类似的资源，所以在设计上 TCP 的关闭是单向的，即无论你是否关闭，我都要关闭了。这很正常，一个远端的机器，不可以阻止我回收属于我的资源，另一端也同样是这样。但是起码的 FIN 要确保发到了对方的操作系统才算负责任（你回不回收是你的事，但是我通知到了），因此不管设计成哪一方发起 FIN，对方都要回复 ACK。

• 205 •

## 5. 连接信息

我们知道 socket 就代表一个通信中的逻辑实体。TCP 虽然只是 socket 的一种用途，但是 socket 也是代表 TCP 连接端点的资源，而且网络中 TCP socket 的数目又远远多于其他协议，所以 TCP 连接的状态直接编码到 socket 状态就是最优的选择。

由于通信是双向的，不可以通过一个数据包告诉对方将自己的状态设为 1 就指望对方也能跟着设为 1，必须要确认对方确实已经设置为 1 了，因为网络是不可靠的。连接的建立、关闭和拥塞控制都是以这个为基础设计的。没有这个前提，所有的机制都没有任何价值。正是通信状态的不可靠，才导致了双方都要为 socket 定义和维护状态。

状态之间的迁移都是因为事件，事件有用户主动发起的，也有收到数据包自动触发的。例如三次握手，对于 client 来说发送了 SYN，自己的 socket 就处于 SYNC\_SENT 状态，收到了 server 的 SYN/ACK 回复，会立即扔出去一个 ACK 包，然后进入 ESTABLISHED 状态。

最复杂的应该是关闭。因为涉及双方 4 次交互，就有 4 个状态。发起方发送了 FIN 之后进入 FIN\_WAIT\_1，收到 ACK 后进入 FIN\_WAIT\_2。此时发起方的资源已经标记为回收，但是之所以还存在就是在等待对方也回收（TCP 是有责任心的协议）。收到 FIN 的一方迅速扔出一个 ACK，然后将自己进入 CLOSE\_WAIT 状态。应用程序检测到 socket 进入了 CLOSE\_WAIT 状态后，需要手动调用 close，该 socket 就可以发送 FIN 并进入 LAST\_ACK 状态，收到 ACK 后 socket 销毁。

有一些特殊情况，比如当发起方发送 FIN 却同时收到了 FIN，发送方发送了 FIN 却收到了 SYNC，连同其正常情况下等待的 SYN/ACK，共有 3 种可能出现的情况。无论其怎么处理，处理完后 socket 都要等待 TIME\_WAIT 时间才销毁 socket (CLOSED)。

在实际的 socket 使用中，你可能会发现处于 TIME\_WAIT 状态的 socket 特别多，处于 SYN\_RCVD 状态的 socket 也特别多。这些可能是攻击，也可能是正常的现象。如果是攻击，这种攻击的作用对象是协议本身，而不是代码的实现。这种漏洞就是协议本身的漏洞，也就是说无论你怎么实现这个协议，这个漏洞永远存在，只能检测和预防而无法归避。例如 TIME\_WAIT 要求主动发起关闭的 socket 要等待一段时间再销毁，目的是让网络中属于该 socket 的数据都传送干净，防止出现重复。而如果频繁发起短连接，迅速打开再迅速关闭，则必然导致处于 TIME\_WAIT 状态的



socket 迅速增多。这是无法避免的，如果要阻止，则违反了协议，也就注定不能由内核来做这件事，只能由用户来选择是否违反协议，内核最多提供机制（`/proc/sys/net/ipv4/`目录下 `tcp_` 打头的所有文件对于深入理解 TCP 都是值得研究的）。

### 8.4.3 TCP 拥塞控制

拥塞控制讨论的是很多个同时存在的 TCP 连接应该怎么规划自己的数据包发送和接收速度，以在彼此之间共享带宽，同时与其他实体的机器公平地竞争带宽，而不是自己全占。

拥塞控制的核心是 AIMD（additive-increase multiplicative-decrease），线性增加乘性减少。为什么不用线性增加线性减少，或者是乘性增加乘性减少呢？因为只有 AIMD 可以收敛聚合使得链路公平。

#### 1. 拥塞检测

##### （1）窗口

由于 sequence 机制具有对网络拥塞的天然的感知能力。因此感知到网络拥塞后如何处理就是应该考虑的问题了。

目前最广泛被接受的思想 and 算法是这样的：网络速度突然发生急剧变化的概率小，拥塞是逐渐发生的（在人们看来拥塞发生得仍然很快，但在机器看来是有个过程的）。而在这个过程中数据逐渐变得不可达。因此，双方应该有学习机制。TCP 留下了用于实现这个的域——窗口。窗口也代表内存接收缓存的大小，缓存本身就有缓冲作用，因此即使是突然变化的网络，在缓存中也有一个过程。正常的通信缓存不会满，但是若一方发送了很多数据（这些数据直到确认前都在发送缓存），却没有被确认，那么可用的发送缓存就越来越小，所以其发送量就开始收缩。同样，接收方知道自己接收缓存的大小，也就知道自己的接收能力。所以其也会及时告知发送方允许对方发送的最高速度。意思就是我现在能一次接收  $N$  个字节，你一次性发我，不要多，也不要少。

这个缓存还有一个很重要的功能就是乱序重组。接收到的数据包放在缓存中，收齐了才会返回给上层，也正是如此，如果接收缓存存在不稳定的网络中，则很容易被填满。

其实接收缓存的作用就是规定最大接收速度和乱序重排，发送缓存的作用就是丢包重传和控制发送速度。

TCP 规定的这个窗口与缓存的大小有关，与确认对方发送数据的 `sequence` 组合，表示接下来可以接收的数据序号区间。窗口只用来告知对方自己的接收能力，不用来表达自己的发送能力。这个发送能力需要根据对方的接收能力和当前自己对信道的估计进行调整。核心的思想就是你不需要在数据包中告诉自己该怎么操作，而应该建议对方如何操作。

## (2) RTT

判断网络是否拥塞，不但可以通过窗口检测拥塞，还可以通过往返时间直接测量。窗口检测是观察，往返时间属于测量。最早的拥塞控制算法 Vegas 就是根据 RTT 来监测和控制的。但是 RTT 不是根据实际的丢包率来计算的，而是根据往返时间计算。而互联网，尤其是无线网，RTT 变大并不意味着不可达或者拥塞，这时使用 Vegas 算法的就开始主动降低自己的速度了（因为它判断网络拥塞了），而其他的基于丢包率的算法则并不减小窗口。导致 Vegas 把可用带宽让给了别人。这种损己利他的做法和 Nagle 一样，是注定要消亡的。

但是 RTT 仍然对拥塞控制有至关重要的作用（除了后来的完全不依赖 RTT 的 cubic 算法），大部分算法都是在收到 ACK 回复的时候才将窗口增加一个 MSS。这就是慢启动的部分（但是增长速度相当快）的原理。

## 2. 拥塞避免

治乱于未乱是最合理的治安方法。害怕发生拥塞首先要想办法避免拥塞，要避免拥塞就要分析拥塞发生的原因，这无疑是网络传输的问题。然而事情并没有那么简单，是什么导致了网络传输的拥塞呢？你可能会不假思索地说是传输的数据太多导致的。然而大多数情况并不是这样，而是技术上的问题。

这里避免拥塞有两个维度的术语：一个是我们不了解细节技术时认为的避免拥塞；另外一个技术上的慢启动算法。整个拥塞控制过程包含了慢启动（窗口指数增长）、拥塞避免过程（窗口线性增长）、发生了拥塞的处理这 3 个过程。

## 3. 带宽节省

避免拥塞的最好方法就是别发那么多数据，但是这对于用户来说是不现实的，毕竟传输数据是网络存在的根本意义。但是内核还是尽可能地从技术上减少用户的

数据，典型的的就是 Nagle 算法。

TCP 传输的数据有两种，即命令和数据。命令的特点是短，而且需要立即响应。数据的特点是长，可以多接收一些再进行响应。针对数据，一般吞吐量较大，立即发送即可，因为上层每次提交到内核要发送的内容很多。但是如果每次都是立即发送命令，则本可以合并在一起发送的数据包会被拆成多个，使得发送同样的数据占用了更多的带宽。为此 Linux 设计了 Nagle 算法。

Nagle 算法规定发送了一个包出去，一直要等到该包的回复才能发送第二个包，在此过程中，数据在发送缓存中累积缓存。如此就可以将尽量多的命令数据合并，从而节省上行带宽。这个算法的初衷是好的，但是效果并不理想，更致命的缺点是它是默认打开的。因为 Nagle 算法对带宽的节省是通过对自己发出的命令的延时进行的，即使超时也得立即发送。但就是这个延时让自己的应用程序感受到系统响应的缓慢，而且这个缓慢还是自己给别人节省上行带宽（发送命令一般占不了自己多少带宽）造成的。这个时候只要自己关闭 Nagle 算法，就会发现自己的传输响应明显加快。典型的应用是 samba，如果关闭 Nagle 算法，TCP 的传输速度一般会提高。

#### 4. 拥塞控制

我们能检测到网络拥塞，也得避免拥塞。现代的所有拥塞避免算法都是基于 4 个核心概念展开的：慢启动、拥塞避免、快速重传、快速恢复。这 4 个基本算法由 Reno 拥塞避免算法首先提出，后来在 TCP NewReno 中又对“快速恢复”算法进行了改进。近些年又出现了选择性应答（selective acknowledgement，即 SACK）算法，还有其他方面的大大小小的改进，成为网络研究的一个热点。Kernel 4.9 TCP BBR 拥塞算法的推出，带来了非常强大的速度提升，但是却是给其他的算法带来了不公平的因素。有个统计规律是单条 TCP 的性能不如多条并行的 TCP 性能，还有一个专门的并行 TCP 库 Parallel socket libraries。

#### 5. 慢开始（慢启动）与拥塞避免

接收方永远通报自己的窗口。发送方根据接收方发来的窗口计算出自己在当前窗口下可以发送的数据序号，例如从 200~500，共 300 个序号，接下来可以随意发送，不需要等待 ACK。假设在接收端回复了下一次窗口为 500，接收确认的是 400 序号，则发送端计算接下来可以随意发送的序号是 400~900，共 500 个字节。如此周而复始。



慢开始算法就是当新建连接时, `cwnd` 初始化为 1 个最大报文段 (MSS) 大小 (或者整数倍, Linux 默认是 65535), 发送端开始按照拥塞窗口大小发送数据, 每当有一个报文段被确认时, `cwnd` 的值就增加 1 个 MSS 的大小。这样 `cwnd` 的值就随着网络往返时间 (Round Trip Time, 即 RTT) 呈指数级增长。事实上慢启动的速度一点也不慢, 只是它的起点比较低而已。

上面的情况是接收方窗口在不断地增大, 这种情况一般会在所有的 TCP 连接建立初期发生。由于两个节点建立 TCP 连接的时候并不知道链路的质量, 所以发送端也不好确认一下子可以随意发送多少数据出去, 所以作为一个对信道的探测, 在 TCP 连接建立的初期, 接收端一般会把窗口设得很小, 然后成倍增大, 这叫作慢启动。增大到一定的值后就会慢速增加 (线性), 是一个逐渐适应的学习过程。从指数增加切换到线性增加的阈值叫作 Slow Start Threshold (SSThresh)。很多算法就在这个值上做文章, 例如动态地变化这个值。

进入的窗口值慢速增加的过程就是拥塞避免的过程, 因为刚开始的时候通常不会发生拥塞, 这个时候慢开始设置的初始窗口太小, 为了快速到达最大速度, 窗口是指数增加的, 但是到了某一个设定的阈值时, 增加窗口就得线性增加, 以防止过快增加导致发生拥塞 (快到极限的时候慢速试探), 这个线性增加窗口的过程就是拥塞避免的过程。这个设定的阈值叫作慢启动门限 (SSThresh)。

在慢启动阶段如果发生了拥塞, 接收方就缩小自己的接收窗口至 1 (或者是系统的默认值 65535), 如此发送方就不会发送那么多的数据了, 还是重新执行慢开始算法。这个算法是 TCP Tahoe。而 TCP Reno 的做法是在慢启动的过程中检测到拥塞时, 把慢启动门限 SSThresh 设置为当前窗口的一半, 所以这样就会强制立即进入拥塞避免阶段。

## 6. 慢开始的缺陷

慢开始与拥塞避免是基于一个常识性的假设: 收到报文了, 说明网络质量有提高的空间, 而发送端要推测有多大的提高空间, 这与 RTT 和当前的窗口大小相关, 所以这个算法是在收到确认后才增加的。而在丢包比较严重的环境下, 比如使用 Wi-Fi 时, 其实带宽是很大的, 只是丢包严重, 这时候这种依据 ACK 确认的机制表现并不好, 增长会比较慢, 并且比较容易被误认为是带宽不够。也就是说发送端无法区别网络拥塞与链路质量 (两者带来的效果都是丢包)。

并且由于是慢开始, 虽然刚开始的时候增长速度很快, 但是基数很小, 所以这

种对于短连接的效果非常差。人们不是试图为短连接采用更好的算法，而是基于现在大家都使用的这种算法想了很多对策。比如同时开多条 TCP 连接，或者是尽量复用同一条连接，但是比如 Web Server 这种服务就无法被正确地满足了。所以你会发现，现在的流量器打开一个网站的时候会使用很多个 TCP 连接去下载不同的资源。

## 7. 快速恢复与快速重传（如何减少窗口的调整）

窗口增大的过程是拥塞避免的过程，窗口减小的过程是拥塞控制的过程。拥塞控制就是在检测到发生了拥塞（或可能发生的拥塞）时，通信双方的反应情况。拥塞窗口的调整可以实现拥塞控制。

### （1）快速重传

除了增大窗口，还有减小窗口的情况，减小窗口的过程一般是剧烈的。在检测到网络中发生了拥塞之后（收不到数据），接收方就会缩小自己的接收窗口，但是怎么缩小各个算法就有所不同了。所谓的 AIMD 算法就是在这里发挥作用的。AIMD 的加性增长就是指冲突避免的过程的窗口线性增大，而乘性减少就是发生了拥塞之后的规避机制（没发生拥塞之前窗口一直在增大，直到物理的缓存内存不够为止）。快速重传就是避免减小窗口而导致窗口波动的算法。

网络是不可靠的，这个不可靠在发送和接收两端的表现是收到重复的包和没有收到包。在 TCP 中，收到重复的包导致的是发送方收到重复的 ACK。其可能认为是网络问题，也可能是接收方一直没有收到某个数据而发送的重复的 ACK。TCP 没有为这种情况设计额外的机制，好让接收方可以在每次发送同样的 ACK 时在数据包上有区别，这就给发送方带来了困扰。传统的收到多个 ACK 时，发送方会认为是网络重传。直到其 TCP 超时机制启动，发现之前发送的包超时没有被回复 ACK，发送方才判断多个 ACK 是自己发送的包接收方没有收到，而不是发送方由于网络原因收到重复的包。快速恢复算法就是在收到 3 个（或 4 个）重复的 ACK 时就判断并给出是因为自己发送的包丢失的结论，从而判断网络发生拥塞。判断发生拥塞就启动重传，但是这时候的重传并不像慢开始算法将窗口回到 1，重现开始执行算法。这时候很大概率只是偶然的网络丢失，所以其只是重发丢失的包，按照原来的速度继续发送，这叫快速重传。这种机制可以显著降低错误收缩窗口的概率。

由此可见，TCP 机制设计得很精巧，堪称异步问题解决的典范。这种设计也是完全建立在物理网络的特性的基础之上的。因为现在的网络是尽力而为的网络，当需求超出其负荷时，其反应是降低服务质量，而不是限制接入数量。如此的网络设

计，就让工作在其上的所有协议都要考虑丢包和重传的问题。

## (2) 快速恢复

快速重传与快速恢复是一个算法，但是由于历史原因，说起来像两个算法的名字。它们分别描述该算法的两个过程：快速恢复与快速重传。快速重传成功了，自然就快速恢复了。

## 8. 其他服务于快速恢复算法的机制

(1) SACK: SACK (选择性确认) 是 TCP 选项，它使得接收方能告诉发送方哪些报文段丢失了，哪些报文段重传了，哪些报文段已经提前收到等信息。

根据这些信息 TCP 就可以只重传那些真正丢失的报文段。需要注意的是，只有收到失序的分组时才可能发送 SACK，TCP 的 ACK 是建立在累积确认的基础上的。也就是说，如果收到的报文段与期望收到的报文段的序号相同就会发送累积的 ACK，SACK 只是针对失序到达的报文段。

(2) D-SACK: 重复的 SACK。RFC2883 中对 SACK 进行了扩展。SACK 中的信息描述的是收到的报文段，这些报文段可能是正常接收的，也可能是重复接收的，通过对 SACK 进行扩展，D-SACK 可以在 SACK 选项中描述它重复收到的报文段。但是需要注意的是，D-SACK 只用于报告接收端收到的最后一个报文与已经接收了的报文的重复部分。

(3) FACK: FACK (提前确认) 算法采取激进策略，将所有 SACK 的未确认区间当作丢失段。虽然这种策略通常带来更佳的网络性能，但是过于激进，因为 SACK 未确认的区间段可能只是发送了重排，而并非丢失。

如果接收端没有收到 3 个以上的数据包，是无法触发重复 ACK 的，只会发送端重传。这种情况只有等到 RTO 才能重传，这就是 TCP 窗口尾丢包的问题，简称 TLP。Google 和 Taobao 都对此做了改进。

## 9. 实际拥塞控制算法的实现

前面是理论基础，但在实现上要考虑不同的情况，设计不同的参数比例。而且网络不断出现新的情况，需要针对性地进行调整。近几年来，随着高带宽延时网络 (High Bandwidth-Delay product network) 的普及，针对提高 TCP 带宽利用率这一点，又涌现出许多新的基于丢包反馈的 TCP 协议的改进，这其中包括 HSTCP、STCP、BIC-TCP、CUBIC 和 H-TCP。现在 CUBIC 是 Linux 使用得最多的拥塞控制算法，



Ubuntu 的默认算法。

总体来说,基于丢包反馈的协议是一种被动式的拥塞控制机制,其依据网络中的丢包事件来做网络拥塞判断。即便当网络中的负载很高时,只要没有产生拥塞丢包,协议就不会主动降低自己的发送速度。这种协议可以最大程度地利用网络剩余带宽,提高吞吐量。然而,由于基于丢包反馈协议在网络近饱和状态下所表现出来的侵略性,一方面大大提高了网络的带宽利用率;但另一方面,对于基于丢包反馈的拥塞控制协议来说,大大提高网络利用率的同时意味着下一次拥塞丢包事件将为期不远了,所以这些协议在提高网络带宽利用率的同时也间接地提高了网络的丢包率,造成整个网络的抖动性加剧。这种算法就相当于明知道网络要饱和了,但是还没有饱和的余量要占用,所以当只有一个人使用时会占尽便宜,当大家都用时就会很快饱和。TCP 拥塞控制是典型的个人利益最大化、集体利益最小化的博弈过程。

BIC-TCP、HSTCP、STCP 等基于丢包反馈的协议在大大提高了自身吞吐率的同时,也严重影响了其他流的吞吐率。基于丢包反馈的协议产生如此低劣的 TCP 友好性的主要原因,在于这些协议算法本身的侵略性拥塞窗口管理机制,这些协议通常认为网络只要没有产生丢包就一定存在多余的带宽,从而不断提高自己的发送速率。其发送速率从时间的宏观角度上来看呈现一种凹形的发展趋势,越接近网络带宽的峰值,发送速率增长得越快。这不仅带来了大量拥塞丢包,同时也恶意吞并了网络中其他共存流的带宽资源,造成整个网络的公平性下降。但是也正是因为没有更高的权威,所以这种算法注定要成为通用的算法。其实笔者也认为,拥塞避免不应该是 server 自己的事情,而应该是路由器 Qos 的事情。笔者倾向于所有的 TCP 都直接不要拥塞控制,将窗口设置为内存支持的最大大小就好了,能不能发送成功、是否拥塞,都交给路由器吧,自己只需要检测丢包率即可。

在技术上,这些算法都是对 Reno 算法在窗口何时以何种幅度增大减小控制策略上的改变。现在主流算法选择的是基于丢包的算法。

### (1) HSTCP (High Speed TCP)

HSTCP (高速传输控制协议)是高速网络中基于 AIMD (加性增长和乘性减少)的一种新的拥塞控制算法,它能在高速度和大时延的网络中更有效地提高网络的吞吐率。它对标准 TCP 拥塞避免算法进行了修改,实现了窗口的快速增长和慢速减少,使得窗口保持在一个足够大的范围,以充分利用带宽,它在高速网络中能够获得比 TCP Reno 高得多的带宽,但是它存在很严重的 RTT 不公平性。公平性指共享同一网络瓶颈的多个流之间占有的网络资源相等。

TCP 发送端通过网络所期望的丢包率来动态调整 HSTCP 拥塞窗口的增量函数。

拥塞避免时的窗口增长的方式为： $cwnd = cwnd + a(cwnd) / cwnd$ 。

丢包后窗口下降的方式为： $cwnd = (1 - b(cwnd)) \times cwnd$ 。

其中， $a(cwnd)$ 和  $b(cwnd)$ 为两个函数，在标准 TCP 中， $a(cwnd)=1$ ， $b(cwnd)=0.5$ ，为了达到 TCP 的友好性，在窗口较低的情况下，也就是说在非 BDP 的网络环境下，HSTCP 采用的是和标准 TCP 相同的  $a$  和  $b$  来保证两者之间的友好性。当窗口较大时（临界值  $LowWindow=38$ ），采取新的  $a$  和  $b$  来达到高吞吐的要求。

### (2) westwood

在无线网络中，TCP westwood 是一种较理想的算法。它的主要思想是通过在发送端不断地检测 ACK 的到达速率来进行带宽估计，当拥塞发生时用带宽估计值来调整拥塞窗口和慢启动阈值，采用 AIAD（Additive Increase and Adaptive Decrease）拥塞控制机制。它不仅提高了无线网络的吞吐量，而且具有良好的公平性和与现行网络的互操作性。其存在的问题是不能很好地区分传输过程中的拥塞丢包和无线丢包，导致拥塞机制频繁调用。

### (3) H-TCP

高性能网络中综合表现比较优秀的算法是 H-TCP，但它有 RTT 不公平性和低带宽不友好性等问题。

### (4) BIC-TCP

BIC-TCP 的缺点首先是抢占性较强，BIC-TCP 的增长函数在小链路带宽时延短的情况下比标准的 TCP 抢占性强，它在探测阶段相当于是重新启动一个慢启动算法，而 TCP 处于稳定后，窗口一直是线性增长的，不会再次执行慢启动的过程。其次，BIC-TCP 的窗口控制阶段分为 binary search increase、max probing，还有  $S_{max}$  和  $S_{min}$  的区分，这几个值增加了在算法上实现的难度，同时也对协议性能的分析模型增加了复杂度。在低 RTT 网络和低速环境中，BIC 可能会过于“积极”，因此人们对 BIC 进行了进一步的改进，即 CUBIC。但是在长肥管道下，BIC 的侵略性是比较合适的。

### (5) CUBIC

CUBIC 在设计上简化了 BIC-TCP 的窗口调整算法，在 BIC-TCP 的窗口调整中会出现一个凹和凸（这里的凹和凸指的是数学意义上凹函数和凸函数）的增长曲线，CUBIC 使用了一个三次函数（即一个立方函数），在三次函数曲线中同样存在凹和凸的部分，该曲线形状和 BIC-TCP 的曲线图十分相似，于是该部分取代 BIC-TCP 的增长曲线。另外，CUBIC 中最关键的点在于它的窗口增长函数仅仅取决于连续的

两次拥塞事件的时间间隔值,从而窗口增长完全独立于网络的延时 RTT,之前讲过的 HSTCP 存在严重的 RTT 不公平性,而 CUBIC 的 RTT 独立性质使得 CUBIC 能够在多条共享瓶颈链路的 TCP 连接之间保持良好的 RTT 公平性。

#### (6) STCP (Scalable TCP)

STCP 算法是由 Tom Kelly 于 2003 年提出的,通过修改 TCP 的发送窗口的大小,以适应高速网络的环境。该算法具有很高的链路利用率和稳定性,但该机制窗口的增加和 RTT 成反比,在一定程度上存在着 RTT 不公平的现象,而且和传统 TCP 流共存时过分占用带宽,其 TCP 友好性也较差。

#### (7) TCP Proportional Rate Reduction

这是压轴的主角,在内核 3.2 版本之后就默认使用这个算法了。Prr 是 RFC 对于快速恢复算法的改进。快恢复是在检测到拥塞发生的时候将发送窗口降低到一半(怎么才算拥塞由另外的算法决定)。

#### (8) PRR

PRR 是最新的 Linux 的默认推荐拥塞算法,之前的拥塞算法是 cubic。但有意思的是,如果在 Linux 中使用了 PRR,那么仍然可以以 cubic 作为默认拥塞算法。因为拥塞算法大致都是一样的,只是在一些参数和细节调整上有区别。Linux 上的 PRR 实现就是对 tcp\_input.c 文件的补充,并不是以模块的形式提供的。

PRR 是一个 RFC 的推荐标准,有推荐的实现方式。内核中对 PRR 的实现也就是 RFC 中推荐的实现方式: PRR-SSRB。

## 8.4.4 TCP 其他的功能特点

### 1. keepalive 保活

TCP keepalive 存在的背景是当很多 server 或 client 在一条 TCP 连接很久没有数据时就会关闭该条连接。如果一方不想 TCP 连接被对方的系统关掉,就可以过一段时间发一条保活数据,这样对方就不能关闭了。

这条保活数据不能含有真实的数据,否则就会影响线上业务(数据会被程序读取到),而 TCP 的标准恰好给了其可行性,如果一方发送一个与前一个 seq 相同的数据包,并且一直 ACK 前一个收到的数据,那么另一方也会一直回复同样的 ACK。这样就可以完成数据的交互了,而不需要传输真实的数据。如果一方没有回复,就



说明连接出现问题了，这就是 TCP 保活的内在原理。

所以 TCP 保活有两个功能：即时探测对端是否出问题和保证让对端不因没有数据传输而断开并关闭连接。/proc/sys/net/ipv4/目录下的 tcp\_keepalive\_time 文件表示 socket 没有数据多久后开始 keepalive 探测，tcp\_keepalive\_intvl 表示两个探测之间的时间间隔，tcp\_keepalive\_probes 表示累计多少个没有收到回复的探测算掉线状态，也可以针对每一个 socket 进行设置。libkeepalive 库是 Linux 上的一个实现 keepalive 的库，完全可以不使用这个库，而使用 setsockopt 调用对 socket 进行 keepalive 的设置。但是如果使用这个库，可以不用做设置，创建的 socket 默认附带 keepalive 的设置，这取决于 libkeepalive 的配置。这个库的原理是在用户层 hook 了 socket 系统调用，为其插入了 setsockopt 选项，代码如下。

```
int keepalive = 1; // 打开 keepalive 属性
int keepidle = 60; // 60 秒没有数据往来 keepalive 激活
int keepinterval = 5; // keepalive 间隔 5 秒
int keepcount = 3; // 探测失败尝试的次数

setsockopt(rs, SOL_SOCKET, SO_KEEPALIVE, (void *)&keepalive,
sizeof(keepalive));
setsockopt(rs, SOL_TCP, TCP_KEEPIDL, (void *)&keepidle,
sizeof(keepidle));
setsockopt(rs, SOL_TCP, TCP_KEEPINTVL, (void *)&keepinterval,
sizeof(keepinterval));
setsockopt(rs, SOL_TCP, TCP_KEEPCNT, (void *)&keepcount,
sizeof(keepcount));
```

## 2. 扫描

TCP 端口扫描有两个目的，一个是发现打开的端口；另一个是发现被防火墙过滤的端口。虽然打开被防火墙过滤的端口，但是从外网扫描不到。扫描的方法是根据 TCP 的几个特性进行操作。判断的标准是端口打开和关闭对不同输入的不同响应（没有防火墙），服务器收到 SYN，如果是打开会回复 SYN/ACK，如果是关闭会回复 RST。收到同时不含 SYN、RST、ACK 的 flag 的数据包打开的端口会不响应，而 close 状态的端口则会响应 RST。

如果对方开启了防火墙，则无法判断端口的打开情况，但起码可以确定哪些端口在防火墙的后面。如果发送只有 ACK 的数据包，端口无论是打开还是关闭都会回复 RST，如果有防火墙数据包则很可能会被丢掉或回复 icmp。而不同的操作系统在



返回 RST 时会设置不同的窗口大小，所以依据这个可以判断一些操作系统的类型。微软与 Linux 的一些种类的扫描的响应不一样，可以因此区别出操作系统。

窗口大小也可以作为扫描的维度。因为新打开的连接会把初始窗口设置为相对固定的值。所以总体的思路是依据 TCP 协议对特定数据包的响应规律进行响应值判断。

### 3. 调优

- txqueue 是内核到驱动的队列大小，将队列调大有助于让上层逻辑不被卡住，但是会增加延时。
- /proc 下的 netdev\_max\_backlog 是指 listen 的时候一个 socket 三次握手完成，但是还没有被用户拿走的最大 socket 数目。这个数越大，内核可以处理越多的并发连接。
- 内核缓存了一些路由信息，但是这些路由信息有可能是不准的。尤其是在高速网络，并且网络质量不稳定的时候。FIB 表的大小的调整、回收速度的调整、flush 等都能在一定程度上优化网络。
- SACK 在高速网络并且竞争小的时候有负面作用，所以这种情况下应该关闭。socket 缓存太小，窗口就不能调高；缓存太大，接收就会过量，甚至可能出现 0 窗口。
- socket buffer size =  $2 \times \text{bandwidth} \times \text{delay}$ ，每个缓存按照这个公式计算。ipv4.tcp\_rmem = MIN DEFAULT MAX，这个 proc 配置是配置内核的自动缓存大小。socket 缓存的配置策略是尽可能配置小的内存，同时保证速度，通常是比较平均的，吞吐量大则分配得就比较大。

## 8.5 网络服务质量与安全性

### 8.5.1 TCP 安全性

当 TCP 连接被同时大量断开时，可以明显地判断为攻击。而这种攻击没有收到防火墙的告警，只是连接断开。抓包可以发现大量的多源地址而非正常流量，例如可能有 RST/SYN 或者是单纯的 ACK。



## 1. Linux 处理 TCP 连接逻辑

这里只提取与我们比较相关的逻辑，其他逻辑对于下文的分析可以忽略。

首先进入 `tcp_v4_rcv`，这个函数会检查数据包的大小和校验码，如果发现错误会直接丢弃。然后检查对应的 `socket` 是否存在在内核的哈希表，如果不存在就返回 RST。这一步的检查就是 TCP 端口扫描的一个原理，可以检查在某个端口是否有服务器在 listen。

进入 `tcp_v4_do_rcv`，对于已经建立的连接，直接进入 `tcp_rcv_established` 函数，接收过程分为快速路径和慢速路径。正常的数据包会进入快速路径，而快速路径只支持单向的连接（只有一方发送数据），所以我们写的应用程序使用长连接来交互数据，大部分事务流量都是进入慢速路径的。还有很多其他情况也会导致事务流进入慢速路径，攻击都是针对慢速路径的。慢速路径就是 RFC793 规定的正常路径。

慢速路径调用 `tcp_validate_incoming` 函数检验输入的正确性，这个函数主要实现 RFC5961 中的 `seq` 值检测和完成盲打的 RST/SYN 挑战。对于 `seq` 值，检查其是否落在窗口内，如果不落在窗口内，就返回 ACK 重复确认。这里的返回 ACK 重复确认就是当前的 ACK 值。也就是说，如果有攻击者一直试探错误的 `seq` 值，服务器将一直回复客户端 ACK 值，而窗口是 0。而如果 `seq` 值正确，又携带 RST 标签，没有实现 RFC5961 标准的系统，就会直接 RST 这条连接，回收所有资源。而无论 ACK 的值是否正确，实现 RFC5961 则会进行一遍挑战。SYN 也是一样的挑战逻辑。

执行 ACK 序列号检查，调用的函数是 `TCP_ACK`。ACK 序列号检查会检查 RFC5961 和 RFC793 中的防止数据注入的逻辑。RFC5961 在 3.0 版本的内核中还没有，在 3.2 之后的版本中就有了。所以，在内核 3.0 版本之前的版本都可以受到这个攻击。

在收到 RST 的时候必须要 `seq` 与预期精确一致，不一致会发送挑战。在已经建立的连接的状态收到 SYN，server 会返回 ACK 挑战，说这个包没有收到，这样如果用户真要重新建立连接就会重新发起 SYN，利用的是 SYN 重传机制。

解决这两种攻击的思路是一致的，这两种攻击的原理也类似。猜测连接的数据注入可以填充连接的窗口内的后部分数据（假设序号是 1234，长度是 11），这样当用户的数据填满前面的时候，server 认为只是顺序不对，需要的下一个数据 `seq` 是 1245。而用户却在拼命地发 1234，因为用户不知道他自己发送 1234 这 11 个字节，这就是 ACK war。双方沟通不在一个频道上，最后只能 RST 断开连接。RFC5961 的防止猜测攻击的方法也很简单，记录一个历史收到的最大窗口，突然跳跃接收





会被拒绝。

## 2. 针对长连接的猜测攻击手法

需要猜测四元组、seq 值、ACK 值,然后就完全拥有了 TCP 连接的注入能力(ACK 的值在很多攻击中不需要猜测)。这两个值只要大致落在窗口范围即可,所以要猜测窗口。这是 in-window 攻击,必须要落在窗口值范围内,所以短连接是很难猜测的,所以本文的攻击对象都是长连接。我们这里针对网页游戏来进行攻击分析。

攻击的方向可以是客户端,也可以是服务端。攻击客户端不容易被发现,并且相对容易,攻击服务端相对难,并且容易被发现。攻击客户端面对的一般是 Windows 操作系统,攻击服务端面对的一般是 Linux 系统,攻击客户端时会发生的情况有很多。

我们从服务端能观测攻击数据,但并不能确定攻击者在客户端做了什么。也不能确定其使用的是什么攻击手法。

## 3. 猜测四元组

对于一个长期攻击,源 IP 比较容易确认,可以通过其他的方法验证源 IP 是否安装了本客户端,所以要猜测的只有源端口。

有的客户端会使用范围固定的源端口,大部分客户端是交给操作系统随机选择的。而操作系统执行这个任务时有很多种方式,Windows 操作系统和 Linux 操作系统也有区别,大致有递增式和随机式方式。并且无论什么情况都有限制值,例如 Windows 操作系统默认在 1025~5000 范围内,大部分用户都不会去修改它。对于攻击者来说,这些客户端的特性是稳定的,而客户端的操作系统也很容易探测出来。尤其是网页游戏,其源端口更是被浏览器进一步限制和区分。

猜测四元组的流程如下。

- 通过其他方法确定使用了客户端的 IP 地址(可以通过客户端的其他特性,或者其他方式完成)。
- 根据对客户端的扫描确认客户端的源端口模式和范围,进而猜测源端口(对一条已经建立连接的 socket 伪造 server 发起 SYN 请求,该客户端会重新建立连接)。

## 4. 窗口大小猜测

对于长时间研究的特定客户端,尤其是能自己在本机安装一份客户端的,窗口



大小很容易猜测。并且猜测的窗口大小不用完全精准，比实际的小也可以（只要由此生成的 seq 值落在窗口内即可）。

很多操作系统都直接设置了默认开始的 32 768 或者 65 535，对于稳定运行的 server 来说，其接收和发送缓存的大小是固定的，所以窗口的大小也在一定范围内。

## 5. 序列号猜测

序列号的取值范围是  $[0, 2^{32}]$ ，但是由于窗口特性，实际的猜测范围是  $2^{32}/\text{window}$ 。比较新的内核都已经修复了这个问题，采用了绝对值验证的方法，使猜测范围回到  $2^{32}$ 。由于现在的带宽都比较大，所以实际的窗口要远大于 10 万，这就更进一步缩小了攻击的范围（RFC5961 之前）。

## 6. 猜测出来后的攻击方法

### （1）攻击内容

- RST 置位，直接断开连接。
- SYN 置位，重新开始链接（由于是伪造的，所以服务器会重新开始不成功）。
- Data: 用坏的数据打乱正常数据，使 server 认为连接混乱，断开连接。

### （2）攻击方法

批量使用在窗口内的一系列 sequence 值（每个值相差一个窗口大小），同时发送（sequence 已经大致猜测出范围）。每次都携带一个 RST 置位（或者使用其他攻击内容）。

假设窗口大小是 2 万，那么只需要 25 万个数据包就可以全覆盖。并且窗口值在现在的网络条件下会比较大，所以可能需要的数据包会更少。

## 7. 接收数据

在内核接受 ESTABLISHED 状态的连接数据时，分为快速路径和慢速路径。默认的情况是进入快速路径，而当出现快速路径实现不了的情况时则进入慢速路径。

进入慢速路径的条件有以下 6 个。

- 收到 0 窗口探测。
- 收到乱序的数据包。
- 收到紧急数据。
- 接收缓存用完了。



- 收到的包中含有坏的 TCP option 或 flag。
- 双方同时在发送数据（快速路径只支持半双工模式）。

所以，你如果想要攻击对方系统使其进入慢速路径，就可以使用以上方法。快速路径主要的速度优化是可以直接把数据拷贝到用户空间，并且直接回复 ACK。这不是标准的 TCP 标准，但确实很快，它是根据数据包从哪个口来的就可以从哪个口出去的思路做的。而标准的流程是来一个数据包，数据要在队列里排队，回复 ACK 的时候还要查路由表。快速通道还可以查看发送列表里有没有要发送的数据，有的话就可以直接发送出去，而不需要经过路由查找。

在 fastpath 和 slowpath 两者之外还有一种 offload，现在的很多网卡都支持，实现了部分 TCP 协议，可以直接回复 ACK。

如果攻击者故意总是发乱序的包或者每次都使用 urg 标志，很快就可以让这条连接的内存队列占满，并且增加 CPU 的负担。如果再加一个 push 标志，而内容却没有发完就可以显著增加 server 的系统调用次数。

client 发送了一个 http 请求，但是把自己设置为 0 窗口，这样 server 的返回就一直发送不出来。

## 8. 预防措施

可以在 TCP 连接上打开 TCP md5 option，但是会降低服务速度。可以使用防火墙来防止扫描，或者使用 RFC793、RFC5961 的内核版本。另外非常重要的一点是健壮的随机值系统，目前内核的随机系统大部分用户足够用了，最直接有效的方法是更新内核。

### 8.5.2 QoS

#### 1. Qos 产生原因

数据包在传输的过程中，在默认情况下对路由器是无差别对待的。路由器认为所有的包都只不过是包，尽力送达即可，不能送达的都扔掉。

网络服务不可能平等。某一些包具备较高的优先级，不被优先扔掉，也不被优先服务。区别不同优先级并提供不同服务的方法叫 QoS（Quality of Service）。





QoS 的产生并不只是因为用户的不同需求。从大背景上看，分立的各个用户对网络不同质量的要求，远远没能到让整个传输网络都升级并且增加计算量的程度。可以有很多更轻量的办法来解决（例如 ISP 接入层次的速度限制和流量控制）。这里的 QoS 并不单指接入，整个传输网络的 QoS 支持代表一种势不可当的需求，这种需求不可能来自于个别的用户。在“三网”融合的过程中，IP 网络展现了巨大的魅力，确切地说从其诞生开始，IP 网络就击败了一切对手（AppleTalk、ISO、Netware、novell 等），从而迅速占领整个因特网和以太网，并且向嵌入式网络、电力网、设备通信协议等专用网络协议渗透。最大的竞争对手是电话网和电视网，这两个网络的特点是具有实时性和多媒体特性。

IP 本身没有野心，使用 IP 的人也没有太大野心，但是作为一个整体，IP 协议栈表现出智能的侵略性。现在，这个整体智能要向最大的对手宣战——电话网和电视网。也只有这种级别的对手，才能让整个 IP 网络有动力去自发地发生变化，且不需要一个整体的架构设计和强大的推动。这就是 QoS 普及的源动力。

## 2. Qos 概述

既然决定了要提供这种服务，接下来就要确定怎么做，最先要解决的问题是要怎样区别不同优先级的数据包。在不同的协议层次通过各自的域能确定本层不同的优先级的值，可以用来描述当前数据包的优先级，也就是说优先级由数据包携带。也可以通过路由器本身根据在路由器上定义的规则（例如哪个目的地址到哪个端口，或端口到端口的优先级配置），来动态检测数据包的优先级，决定为其提供的服务。

QoS 分为两类：集成式服务和区分服务。集成式服务就是路由器根据规则决策，指定决策规则的方法是通过一个叫作 RSVP 带宽预留的协议。用户使用这个协议与路由器通信，路由器会将用户指定的带宽预留给用户。区分服务就是数据包携带优先级，由于数据包有很多个层次，在哪一层携带都是可行的，由于不同层对网络具有不同的认知能力，所以不同层携带的优先级编码注定只能代表本层的认知。例如 IP 层的 TOS（DSCP）代表了 IP 数据包的优先级（从源 IP 地址到目的 IP 地址）。加上 TCP/UDP 层的端口的概念就变成了一个 socket 到另一个 socket 这个抽象的数据流的优先级的概念。路由器来决定使用数据包或者数据流的概念来区分不同的优先级。

区分了不同的优先级后，要决定怎么对待不同优先级的数据包或流。典型的对待方式是保证速度、保证质量、限速（整形）、优先丢弃、最省钱方式等，总体分为



两大类：整形和策略。整形是速度控制，策略是丢弃决定。路由器能做的，也只能如此。策略算法有很多种，整形算法也有很多种。计算机科学就是通过这种划分与专门治理从而快速发展的。将不同优先级的数据或数据流放入不同的队列，不同队列采取不同的对待方式，分阶段完成 QoS。

### 3. Qos 的优先级划分

在编程的世界里，数据结构是算法的基础。在现实世界里，领土划分是国际关系的基础，物理属性是根本。划分方式能在很大程度上影响功能的使用，甚至划分本身也是根据使用而制定的。

前面说过，划分可以在不同的层次。常见的是 MAC 802.1p 和 IP 层的 TOS，还有 ATM 的服务类别。但是目前在现网使用过程中，最终还是 IP 层的 TOS 完全胜出，所以这里只讨论 TOS。

TOS 是 IP 协议定义的 IP 头部的一个域。不是由于这个域的存在才可以做 QoS，而是由 QoS 的需求才设计了这个域。这个域虽然有 8 个字节，但是这 8 个字节的定义却是逐步确定的。在 RFC 791 中只使用了 6 个比特用于定义优先级，而在 RFC 2474 (DSCP) 中使用了全部的 8 个比特，详细定义了各种服务划分。

前面说的 TOS 是一种思想，叫作 QoS，但是实际完整的 QoS 需要对网络的改造太大，所以因特网上几乎没有真正完整的 QoS 实现。有一种比较简单的 QoS 叫作 COS (class of service)，这不是针对服务质量的，而是针对服务种类的。实质上 COS 只区分优先级，总是优先发送高优先级的数据，低优先级的缓存待空闲时发送，或者在缓存不足时直接丢弃。而 QoS 保证的是质量，不是优先级。这个质量主要是指为实时的应用留出带宽，是一种预留，而不是数据包来了以后的判断。使用 QoS 预留带宽必须要首先说明预留的量，其他用户的丢包率控制在可接受的水平内。QoS 保证的是质量，是通过服务的类型来决定的而不是通过优先级来确定哪个包优先通过的。CoS 是优先级队列形式，已在大量通信和连网协议中使用，也是一种基于应用类型（语音、视频、文件传输、事务处理）、用户类型（CEO、秘书）或其他设置对数据分组区分类别并区分优先级次序的方法。

### 4. QoS 工具

QoS 是一个概念，内核中也在为其设计，但是实际的落地需要有工具和系统，在 Linux 下 QoS 就可以直接说成是流量控制。流量控制常用的软件是 tc 和 tcng。tcng

是流量控制领域的编译器，但是已经找不到 Ubuntu16 这个软件包了，但是 tc 仍然存在。

流量控制就是决定什么样的数据包在什么口以什么样的速度接收，并且什么样的数据包在什么口以什么样的速度发送。流量控制的核心组件有以下 7 个。

- 队列 (queue): 区分的流量需要放在不同的队列。
- 整流器 (shaping): 队列的好处是放进去和出来的速度不用一样，只要保证不满就行。整流器就是队列进出的速度控制组件。主要控制出的速度，让出的速度按照我们希望的那样，这就可以实现以特定的速度输出的目的了。
- 调度器 (scheduler): 在入队列的时候，需要决定不同流入队列的顺序和优先级。
- 分类器 (classifier): 决定什么样的数据包进什么队列。
- 策略器 (policer): 队列中数据包出的时候，为了满足一定的要求，而对满足和不满足的数据包采取的措施（丢弃、重新分类、通过等）。
- 丢弃器 (dropping): 就是实际采取丢弃的组件。
- 标记器 (marking): 会给数据包打上标记，共后续使用。也是策略器所采取的行动。

这些组件都是如同积木一样搭建或者互相包含的。这些定义都比较重量级，但是实际的功能都比较轻量级。

数据包走入 tc 的时候只有一个入口，但要选择进入多个出口中的某一个，这个出口的选择就是调度器的工作。调度器本身也是有入口和出口的。典型最常用的调度器是 qdisc，它的入口是 ingress qdisc，出口是 egress qdisc。egress 用得比较多，其又叫作 root qdisc。在 ingress qdisc 中，一般向上挂载 policer 来做流量控制。在 egress qdisc 中，一般向上挂载分类器 (classifier) 和过滤器 (filter)，由于核心的工作是分类，而只有 egress disc 可以挂载分类器，所以其重要性比较高。代码如下。

```
tc class ls dev eth0 //显示当前 eth0 的分类器
tc filter ls dev eth0 //显示当前 eth0 的过滤器
```

分类器本身是有等级的。一个 class 可以包含子 class，只有叶子 class 有 FIFO，所有的匹配都最终匹配到叶子 class，然后进入叶子 class 的 FIFO。不是叶子 class 的分类器是内部分类器，都不带 FIFO，匹配到了 QoS 规则只会继续向下传递去匹配子 class。

在 class 的后面可以添加 filter，class 虽然匹配到了规则，filter 仍然可以对匹配



的结果进行过滤。内核的 CONFIG\_NET\_SCHED 系列选项是用来配置调度器的, CONFIG\_NET\_CLS 系列选项是用来配置分类器的。

Qdisc 本身有很多种, 比如 HTB、HFSC、PRIO、CBQ 等不同的分类器。至于 tc 命令的使用, 则必须详尽地研究 tc 背后的实现原理才能使用。tc 不是简单的命令, 而是一个复杂系统的前端, 如 ip 命令一样。

### 8.5.3 NAT

#### 1. NAT 概述

NAT 是一种将一个 IP 域与另外一个 IP 域映射起来的方法。其目的是向主机提供透明的路由。传统上 NAT 是用来将一个独立私有的、没有注册的 IP 域与另外一个外部带有全球唯一注册地址的 IP 域映射。

NAT 的作用是绑定一个可以在外网通信的地址和一个内网私有的地址。这种绑定可以查表式地静态绑定(一直不变), 也可以根据需求和可用资源动态产生动态绑定。静态绑定适用于内网机器较少的情况, 或者需要稳定的对外 IP 地址的情况。由于动态绑定可以回收和重新分配地址资源, 适用于机器较多、资源较紧张的情况。

NAT 分为 Traditional NAT、Bi-directional NAT、Twice NAT、Multi-homed NAT。其中 Traditional NAT 较为常见, 其又分为 Basic NAT 和 NATPT。

Basic NAT 是一种将内网的 IP 地址与外网的 IP 地址关联绑定的技术。只作用于 IP 层。NAT 设备(一般为路由器)持有多个全局 IP 地址, 将内网的 IP 地址与全局的 IP 地址绑定映射, 可以采用静态映射, 或根据需求动态映射。

NAPT 的运作条件是 NAT 设备(一般为路由器)只有一个 IP 地址, 而内网有很多私有地址需要区别的外部通信。这时 NAT 设备利用传输层的端口机制将内网的 socket (IP: port) 唯一映射为外网的 IP: port。这种方法可行的原因是, 在大部分情况下端口有很多空余。此时内网所有用户用到的端口总数就是 NAT 设备需要用到的端口数。当端口资源耗尽时将无法正常映射, 所以这种方法在较大的网络中(超过 6 万台机器)可能出现问题。

锥形 NAT 是针对每个内网的 IP:PORT, 无论该地址向多少个外网发起多少个连接, NAT 设备只给其固定分配一个外网可用的 IP:PORT 地址。而对称的 NAT 则是

在内网同个 IP:PORT 发起不同连接时, 给其分配不同的外网可见的 IP:PORT 地址, 并且只有在内网首先与外网通信后, 外网才可与内网通信, 而锥形 NAT 有多种实现方式。锥形 NAT 分为以下几类。

- 全双工锥形 NAT: 一旦一个内网的 IP:PORT 与 NAT 设备的外网可用 IP:PORT 绑定, 外部的所有用户只要知道了该外网绑定的地址, 就可以通过与这个地址的通信与内网建立连接。
- 受限制锥形 NAT: 与全双工锥形 NAT 不同的是, 当内部地址与外部地址发起 session 时, NAT 设备会记录该外部的 IP 地址, 只有被 NAT 记录的外部 IP 地址 (不限端口) 才可以把数据传给映射的外网可用地址将数据传进内网的地址, 即只有内网与外网地址通信过, 该外网才可与内网通信。
- 端口受限锥形 NAT: 在受限制的锥形 NAT 的基础上, 不只是限制通信过的外网 IP 地址, 而且必须是通信过的 IP:PORT 地址, 即只有之前内网给外网的 IP:PORT 发送过数据, 该 IP:PORT 地址才可以发送数据给内网。

Basic NAT 仅用于有足够多的全球唯一的 IP 地址的情况下, 因为它要为内网的所有有通信需求的机器分配一个不重复的 (唯一) 外网 IP 地址。当 NAT 设备没有足够的外网 IP 地址时, 或者只有一个外网 IP 地址时, 就该启用 NAPT。通过将 IP:port 映射完成地址复用。这么操作产生的问题是不但会修改 IP 头部, 还会修改传输层的头部, 导致校验全部重新计算。并且要用到传输层头部的应用 (如 FTP), 都需要重新修改以适应地址的变换, 针对不同上层应用的 NAT 修改程序称为 ALG。

NAT 有 4 种类型, 大部分的家用户路由器的测试结果都如图 8-13 所示。

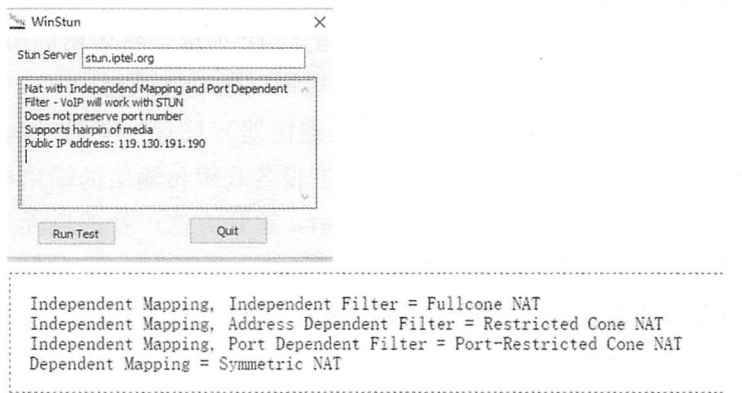


图 8-13

也就是图 8-13 所表示的端口受限, 要求发送 ip:port, 目的 ip:port 必须要互相对应, 即路由器打开的端口只接受两边 IP 端口都吻合的数据包穿透 (也就是一个 TCP 连接)。

由于使用 NAT, 使 IPSec 失效, 此时就必须考虑安全问题。由于 NAT 具有动态绑定的特性, 可以使内网主机不容易受到攻击, 并且经常与防火墙共同使用。锥形 NAT 具有解决 UDP 天然不安全机制的能力。由于 UDP 无法跟踪发送者, 导致主机一旦暴露了自己的地址, 就会收到任意的 UDP 包, 只要目的地址正确。锥形 NAT 通过受限和端口受限两种级别来过滤源地址, 保证收到的 UDP 数据包是自己想要收到的。但是简单的机制具有盲目性, 如果内网想要接受不受限制的 UDP 访问, 比如运行服务器程序, 限制就会阻碍服务器的正常运行。所以, 全双工 NAT 这种不受限制的访问适用于 NAT 内网运行服务器程序的情况。

在图 8-14 所示的网络拓扑中, 两个主机都位于 NAT 网关之后, 两个路由器后面的 host 1 与 host 2 为 P2P 的两个节点, server 为服务器, 其作用就是“打洞”, 让 host 1 与 host 2 建立多条 TCP 连接。由于在极端情况下 (端口受限) host 1 与 host 2 之间必须互相在特定的端口发送过数据才能互相通信。而这之前是没有发过的, 甚至互不知晓, 服务器的责任就是让 host 1 与 host 2 互相知道对方的 IP:PORT 地址。知道对方的 IP:PORT 地址之后, host 1 的 IP:PORT 与 host 2 的 ip:port 互相发送信息。此时各自的 NAT 就会记录通信记录, 虽然第一次互相发送的数据不能成功到达后端主机, 但之后的通信就可以建立。此时完成打洞, host 1 与 host 2 可以自由通信协商, 以建立后续的多条连接。

可以利用 SSH 的端口转发快速地突破 nat, 命令如下。

```
ssh -o ServerAliveInterval=20 -f -N -R 1234:127.0.0.1:1234  
sshtest@1.2.3.4 -p 22
```

这个命令在内网中执行可以直接在 IP 地址是 1.2.3.4 的自己的外网主机上打开 1234 端口, 这样任何访问本机 1234 端口的程序都会被导入到 1.2.3.4:1234 端口去。这相当于在一个内网中开了一个洞。

## 2. ALG

FTP 的 PORT 命令和 PASV 命令都用到 IP 地址和 TCP 端口, 而 NAT 会修改这两个地址, FTPALG 是运行在 NAT 设备上的, 其目的就是检测这种情况并做合适的转换。NAT 拓扑图, 如图 8-14 所示。



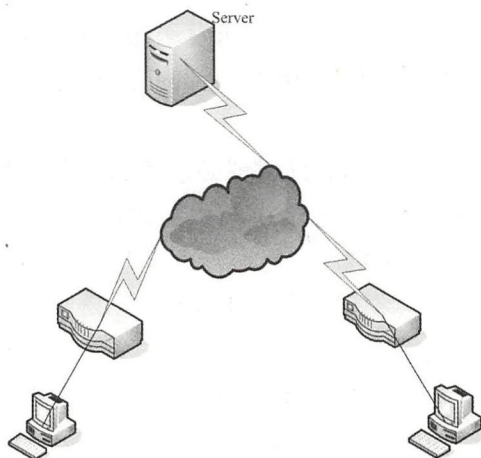


图 8-14

FTP 主动连接的数据传输过程是：由之前已经建立的控制连接发送 `port` 命令，命令的参数是数据连接的 IP:port 地址，服务器在接收到这个地址之后，就向这个地址启动数据连接，开始发送数据。如果 `port` 命令所携带的 IP 和 port 是错误的（没有 FTP ALG），数据连接不能正常建立。此时与服务端有没有 FTP ALG 无关。

在被动连接时，客户端需要用 `PASV` 命令向服务端请求一个 IP:PORT 地址，服务端通过 `response` 通知目的地址自己已经打开供数据连接使用的 IP:PORT 地址，用户端的 NAT 不会翻译这个地址，因为它是外部的源地址。对于 NAT 来说，外部数据只会翻译目标地址，所以 `PASV` 数据包可以正常到达。所以 FTP 的被动模式在用户端不需要 FTP ALG（若服务端也工作在 NAT 之后，则服务端的 NAT 就需要 FTP ALG）。

综上所述，在 FTP 的主动模式下，用户端需要 FTP ALG。被动模式下，服务端需要 FTP ALG。其他常见的 ALG 还有 DNS-ALG、ICMP、GRE、PPTP、ESP 等。

## 9

## 第 9 章

## 总线与设备变动

内核中重要的总线存在一个重要的纵向系统，就是 USB 到 PCI 的过程，并且中间涉及的 SCSI 子系统能够有效地帮助我们理解后面的存储章节的内容，所以本章重点贯穿 USB 到 PCI 的总线过程，中间同时引出 SCSI。通篇研究/sys/bus/目录能够有效地帮助提高对 Linux 总线系统的掌握能力。

## 9.1 PCI

PCI 是一种总线，PCI-e 是 PCI 的升级版，在 Linux 的软件系统中统一都在 driver/pci 下。既然是一种总线，在物理上就包括总线部分和支持该总线的设备。如果没有设备，PCI 总线的存在也没有意义，没有 PCI 总线的支持，PCI 设备就无法发挥作用（就得使用其他总线）。

众多设备本身没有从属关系，如果任由它们任意接入系统，就会在 CPU 和内存资源方面产生竞争，因为哪个设备都希望首先并且获取最多的系统资源。网络接入里常用的 CSMA/DA 机制就是为了解决这种问题而存在的。但是当网络速度非常快的时候，这种竞争随机退避的算法对时间的浪费就会极大地影响性能。所以一般的高速网络要么是协调的，要么传输是物理上独立的信道。对于计算机系统来说，独立的物理信道由于成本问题实现起来不现实，但是在一条物理信道上协调通信却是

可行的，因为计算机系统的核心就是自己本身的处理能力。

这就是总线存在的原因。总线有很多种，为什么总线有好有坏呢？也就是说各个总线之间是有区别的。因为总线的本质是调度资源，而调度分配资源的算法各异，所以评价调度资源的效率就是最主要的评价总线的方法。但是除此之外，成本、可扩展性、可热插拔机制等辅助功能（通常是方便使用的）的支持程度，也是一个总线是否被广泛接受的重要因素。市场是最好的决策者，各个总线在竞争中很多都被淘汰了，存活下来的最耀眼的就是 PCI，耀眼到以至于 Linux 内核认为几乎所有的设备都是挂载到 PCI 上的，甚至所有外部总线都是挂载到 PCI 总线上的。

上面说的外部总线，例如 USB 总线的热插拔能力，就是外部总线的一种能力，外部总线通常的特点是相对核心内部总线速度较慢，但是在可扩展性和热插播能力上做得更好。

## 1. Linux PCI

PCI 是总线，所以驱动必然要分为两部分：总线部分和接入总线的设备部分。总线部分描述和实现的就是 PCI 总线的规范，而接入总线的设备驱动描述的是接入设备的行为。按照惯例，所有驱动都要注册到 PCI 总线部分的驱动上，以方便总线驱动完成枚举、发现、省电、错误处理等统一的总线操作，并且以总线统一的认知方式提供设备的信息，例如商家、设备类型等。

驱动并不等于设备。驱动存在于代码的软件模块中，设备是在物理上存在的，驱动是为设备服务的。当设备插入的时候（或在插入之前）驱动已经存在于系统中了，否则设备不可被识别。当设备被移除的时候，驱动也不会消失（可以稍后卸载）。驱动早于设备识别启动，晚于设备移除而移除内存（可以不移除）。因为无论是设备的添加还是移除都需要驱动里对应的函数被执行。PCI 设备也不例外，任何一种 PCI 设备在插入时，代表该种设备的驱动的 probe 函数都将被执行。移除时，执行驱动的退出函数。一个驱动对应一种设备，一个设备通常对应一种驱动（但复合的设备可能对应多种驱动）。

笔者有两个网卡，但是两个网卡通用一个驱动。在这个驱动里的 bind 和 unbind 文件可以对其写入来决定是绑定还是解绑这个网卡设备。如图 9-1 所示，给出了绑定和解绑的操作。



```

root@ubuntu:/sys/bus# cd pci
root@ubuntu:/sys/bus/pci# ls
drivers autoprobe drivers_probe rescn resource_alignment uevent
root@ubuntu:/sys/bus/pci# cd devices/
root@ubuntu:/sys/bus/pci/devices# ls
0000:00:00.0 0000:00:1a.0 0000:00:1c.2 0000:00:1f.0 0000:00:1f.3 0000:01:00.1 0000:03:00.0 0000:04:00.0 0000:06:00.0
0000:00:14.0 0000:00:1c.0 0000:00:1d.0 0000:00:1f.2 0000:01:00.0 0000:02:00.0 0000:03:01.0 0000:05:00.0
root@ubuntu:/sys/bus/pci/devices# lspci|grep net
01:00.0 Ethernet controller: Broadcom Corporation NetXtreme BCM5720 Gigabit Ethernet PCIe
01:00.1 Ethernet controller: Broadcom Corporation NetXtreme BCM5720 Gigabit Ethernet PCIe
root@ubuntu:/sys/bus/pci/devices# cd 0000:01:00.0
root@ubuntu:/sys/bus/pci/devices/0000:01:00.0# ls
broken_parity_status consistent_dma_mask_bits driver index local_cpulist msi_bus remove reset resource0_uc resource4 subsystem uevent
class device dma_mask_bits firmware_node irq local_cpus numa_node rescn resource0 resource2 resource4_uc resource4 subsystem_device vendor
config
root@ubuntu:/sys/bus/pci/devices/0000:01:00.0# ll driver
lrwxrwxrwx 1 root root 0 Nov 20 2014 driver ->
root@ubuntu:/sys/bus/pci/devices/0000:01:00.0# cd ../0000:01:00.1
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1# ll driver
lrwxrwxrwx 1 root root 0 Nov 20 2014 driver ->
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1# cd driver
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver# ls
0000:01:00.0 0000:01:00.1 bind module new_id remove_id uevent unbind
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver# echo -n "0000:01:00.1" >unbind
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver# ls
0000:01:00.0 bind module new_id remove_id uevent unbind
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver# echo -n "0000:01:00.1" >bind
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver# ls
0000:01:00.0 0000:01:00.1 bind module new_id remove_id uevent unbind
root@ubuntu:/sys/bus/pci/devices/0000:01:00.1/driver#

```

图 9-1

一个 PCI 设备可能内含多种设备，该种复合 PCI 设备的驱动就有理由不使用总线的注册流程，而是自己去扫描发现属于自己能够驱动的设备。当然，自己扫描也是利用 PCI 总线的数据接口与未知设备通信，通信方式是统一的，如果对象设备是驱动所希望的，就会回复所期望的回复，而其他设备则不认识或回复得不正确。也就是说这两种驱动的设备发现逻辑，一种是由总线驱动调度的，另一种是由设备驱动调度的，但是两者都是调用总线的传输接口。

## 2. PCI 设备的初始化

上面讲述了 PCI 设备的发现，发现后要进行初始化，要时刻铭记初始化由物理设备驱动和总线驱动的软件代码实现。

那么这里就涉及一个驱动代码和总线代码的交互问题。驱动代码想要做具体的事情都要调用总线代码，而驱动代码决定如何去调用。这就相当于编程时要使用一个库中函数，彼此是调用关系。但这里的总线代码有自己的逻辑，即可以看出是两个独立的线程实体。总线驱动和设备驱动同时运行，设备驱动依赖于总线驱动而正常工作。

## 3. PCI 地址空间

PCI 总线协议不仅规定了总线驱动的功能，还规定了想要在 PCI 设备上通信的设备所具备的物理条件，任何一个宣称支持 PCI 的设备都必须符合 PCI 协议的规定。其中最重要的就是 PCI 地址空间。

地址空间是一个会出现在每个计算机系统中的词语。因为所有的处理器对外只看得见地址和地址里面存储的内容。内存数据、设备控制寄存器、设备缓存，甚至

磁盘数据等都是通过将自己映射到处理器可见的地址空间中才得以被处理器发现并使用的。PCI 作为一个高速总线，其总线本身的寄存器就位于 CPU 的地址空间内。对于特定的 CPU，以 x86 为例，CPU 能看见两个地址空间，即内存空间和 I/O 空间，PCI 规定在 x86 结构下，PCI 的放入入口位于 IO 空间。但是 IO 空间资源非常有限，所以 PCI 只占用了两个地址（共 8 字节）。

由于 PCI 总线只占用了 CPU 的 8 个字节的空間，而 PCI 的功能又很复杂，8 个字节既得读又得写。所以 PCI 精细地将这 64 个位分解为特定的域，通过不同的组合访问不同的设备，并且以双工的方式将两个地址字分为一个地址（CONFIG\_ADDRESS）、一个数据（CONFIG\_DATA）。例如地址字有的域代表总线编号；有的代表设备编号；有的代表功能编号；有的代表前面域定位到的设备内部的寄存器的编号。这样前面刚定义的唯一索引到的设备内部的寄存器通过 CONFIG\_DATA 暴露出来就可以读取和写入了。举个例子，代码如下。

```
echo 0 > /sys/bus/pci/slot/$N/power //关闭某个 pci slot 的电源
```

#### 4. PCI 设备配置空间

前面说了寄存器编号，这是每个 PCI 设备都要提供的寄存器。PCI 规定了 PCI 设备所需要提供的寄存器的数目和功能，如此上层的 PCI 总线就可以对任何种类的设备设置已知的寄存器编号，以获取和设置该设备的特定功能和信息。这些被 PCI 协议预定义的每个设备都必须实现的寄存器叫作 PCI 配置空间。PCI 规定该空间总长度为 256 个字节，头部是 64 个字节。因此写入 CONFIG\_ADDRESS 的寄存器编号的取值范围是 0~63。

这 64 个配置空间寄存器也被 PCI 协议预定义。这 64 个寄存器中最重要的有两个，一个是描述设备信息的 Device ID 和 Vendor ID（就是设备号和厂商号，Vendor ID 需要向 PCI 协会申请）；另外一个是指针。

#### 5. 设备配置空间的访问方法

可以通过寄存器直接访问 PCI 寄存器，但这是 x86 的访问方式，mips 或 arm 的访问方式另有规定。由于在不同的平台访问方式不同，所以 Linux 就有了抽象的接口（甚至可以通过 BIOS 编程接口访问）。但是 Linux 的抽象接口必定与 Linux 内部的抽象相关，例如 `pci(read|write)_config(byte|word|dword)` 函数可以直接读取一个设备的配置空间的寄存器，然而其需要提供的参数是 `pci_dev` 结构体，这是与硬件

无关的编程思想。

sys 文件系统还直接给出了这部分地址空间的内容，`/sys/bus/pci/devices/0000:00:01.0/config` 文件就是 PCI 的 256 个字节的配置空间。

## 6. PCI 设备内存缓存数据空间

配置空间的地址指针是用来做什么的呢？我们刚说了 CPU 配置和访问 PCI 设备信息通过两个 I/O 地址空间字进行，但是 PCI 是个高速总线，如此快的速度，两个字的大小不可能完成全部的数据交互功能。但这两个字是配置和获取信息使用的，并不是数据传输的空间。而数据想要传输，必须使用内存，要让 CPU 看见，也必须有内存空间的地址。这个地址在每个设备都有一份，并且是不同的，因为每个设备都要传输数据，也需要缓存数据。因此定义这个地址的最好位置就是在 PCI 设备配置空间的寄存器中，所以在配置空间中就定义了地址。由于每个设备不可能固定地确定自己所要使用的内存地址（否则会冲突），应该由操作系统根据当前内存的使用情况来动态分配。因此配置空间的数据地址寄存器应该由操作系统，也就是 PCI 总线驱动写入。

## 7. 动态确定 PCI 设备的数据地址

这个机制也是 PCI 总线协议规定的。由于每个 PCI 设备所需要的内存数据空间不是一样的，也就是说每个设备不能要求地址的具体位置在哪儿，但是必须在配置空间中告诉系统要多大的内存。这两步是通过同一个寄存器完成的（32 位基地址寄存器）。一个设备会将该寄存器的部分位设为可写，部分位设为只读。当系统上电的时候会向全部的位写入 1，由于部分位不可写，其值就为 0，部分位可写，其值就为 1。如此操作系统再读取就会得到一个设备相关的值，该值就表示需要的空间大小。比如该设备需要 64KB 的地址空间，这个值就是 `0xFFFF0000`。系统得到设备需要的空间大小后，就分配所需大小的内存，并将其分配得到的地址写入该寄存器的可写部分中。这样 PCI 设备就知道它所拥有的内存地址空间了。而内核中的其他组件就可以通过 `CONFIG_ADDRESS` 和 `CONFIG_DATA` 两个寄存器读取该设备的配置空间来获得该设备的内存数据地址了。

这个地址位于 PCI 设备硬件上的最大好处就是方便 DMA，由于设备直接可见数据地址，所以可以不经 CPU，直接使用 DMA 引擎将数据传输到内存中。



## 8. 初始化流程

上面描述的 PCI 设备内存基地址的确定的步骤是设备初始化的一部分。完整的初始化流程如下。

(1) 启动 PCI 设备。要使用一个 PCI 设备必须经过总线驱动的同意，否则总线驱动不予传达后续操作命令。

(2) 初始化 PCI 设备的内存数据空间、基地址寄存器和 DMA。

(3) 向中断子系统申请中断号。数据 DMA 传输后要通知内核，通知内核要使用中断号来让内核知道。

设备移除的关闭流程与初始化流程相反。

## 9. PCI 驱动职责与 PCI 总线驱动职责

总线驱动是固定的，设备驱动要不同的设备商家使用总线驱动来实现。而 PCI 总线中规定了很多操作，这些操作并不是每个设备驱动都要实现的，正因为是总线规定的，所以具有通用性，很多逻辑就实现在了 PCI 总线驱动中，所以 PCI 设备的驱动开发者就得明白什么不需要自己实现，什么需要自己实现。如设备 ID，总线驱动中不可能全部包含，所以各个驱动就需要在总线驱动不包含所需设备 ID 的情况下自己定义在驱动中。

另外，虽然总线驱动实现了 PCI 设备的发现和配置，怎么使用这个配置也是由总线规定的，但是实际的使用者却是驱动。例如 PCI 总线驱动配置了 PCI 设备的内存数据的地址，但是设备驱动在向这个地址写入数据的时候必须得了解这个地址是否是内存地址，而驱动要写入的数据是要写到设备中的。写入内存地址后，DMA 并不一定立即启动将数据传输到设备，就算启动也需要时间。因此如果设备驱动在写入内存后要读，需要等待一段时间或者强制地将数据立即刷到设备上。

## 10. PCI 中断系统：MSI

MSI 全称为 Message Signaled Interrupts。在 PC 机的物理系统上，中断是通过引脚的高低电平实现的，一般的 CPU 只有很少的中断输入，为了区别更多的中断，通常外置中断芯片，中断芯片向外提供很多中断引脚，通过查询中断芯片的寄存器获得是哪个设备的中断。先进一些的系统在 CPU 内部就有分级的中断标识，但所有这些都是通过引脚高低电平变化实现的。内核定位中断类型是通过树形的寄存器组织追踪哪个置位来确定的。

Linux 内核中抽象了这种硬件树形中断架构，因为不同的芯片树的组织不一样，简单的几个扁平的寄存器就可以代表所有中断，甚至中断类型不需要使用寄存器，而是使用中断函数去查询并确定是哪个设备的中断，比较难处理的是多个 CPU 分享组织不同的中断。组织有不确定性，但区分不同中断的需求是确定的，Linux 内核提供了以中断号为核心的中断系统。每个中断对于 Linux 来说都是某个中断号（例如 32 号中断）。如果有中断，就调用与该中断号关联的中断处理程序执行，一个中断号可以挂载多个中断处理程序，以方便多个驱动共享同样的中断号。

PCI 协议为 PCI 设备定义了新的中断方式，虽然如此，其上层也使用操作系统固定的中断方式。也就是说任何一个 PCI 设备发生了中断，PCI 总线对应的中断引脚还是会起作用的，中断号会被激发，对应的中断处理函数也会被调用。所不同的是，PCI 是总线，对于 CPU 来说虽然只是一个 PCI 中断，但实际上可能是总线上任何一个设备的中断信号。PCI 协议定义了一种协议来区别是哪个 PCI 设备的中断，这种协议叫作 MSI（或升级版的 MSI-X）。

MSI 的原理是任何一个 PCI 设备发生中断，其向 PCI 总线的驱动程序发送一个信号消息，该消息代表了具体的中断信息。这种机制会在内存中模拟出类似传统的中断寄存器，只是大小不受限于一个寄存器的大小。因此 MSI 可以支持 32 个中断源，MSI-X 可以支持 2048 个中断源。MSI 的 32 个中断源在内存中必须连续分配，而 MSI-X 则不需要。MSI 可能受限于单个 CPU，而 MSI-X 可以跨 CPU 分配。可以看出 MSI-X 是对 MSI 的升级。

MSI 的存在，使得触发中断的时机可以由设备掌控（例如让数据传送完毕再触发中断），并且可以让一个 PCI 设备触发多种中断（这在传统架构中是不可以的）。如果没有这种机制，处理中断的步骤将会是 PCI 总线的中断引脚被触发，CPU 只会知道是 PCI 总线上的某一个设备发生了中断，然后调用中断处理程序，遍历 PCI 总线上的所有设备，直到找到发出中断的设备。由于 PCI 响应速度快，发生中断很可能是并发的，如此的响应方式就会产生很多问题（例如饥饿）。

当然，Linux 允许你关闭 PCI 的 MSI 功能（PCI 协议也是允许的），关闭之后就采用传统的中断处理方式去遍历寻找。这么做终端处理效率会显著降低，但是考虑到不是所有的 PCI 设备都支持 MSI，所以这种支持是有必要的。

## 11. PCI-E 错误处理

一个总线协议标准一定会定义错误处理与恢复，相对应的一个总线的软件驱动也必须实现这种错误处理机制。在 Linux 中这个机制实现的名字叫 PCI Express Advanced Error，简称 PCI-E。

PCI-E 定义了两种错误处理方式，一种是所有设备都支持的 Baseline Capability，提供最小支持，但是要求所有 PCI 设备都支持可以汇报；另一种是扩展的 AER (Reporting (AER) Driver) 机制，其实就是提供更多的错误信息，方便前端用户调试和恢复。AER 驱动就是收集这种扩展信息，并且提供给用户的机制。

## 12. PCI 设备用户空间视图

用户可以用 `lspci` 命令来查看当前的 PCI 设备的信息（使用 `-vv` 查看详细内容），如图 9-2 所示。

```
root@ubuntu:/sys/bus/pci/devices/0000:00:01.0# lspci
00:00.0 Host bridge: Intel Corporation X58 I/O Hub to ESI Port (rev 13)
00:01.0 PCI bridge: Intel Corporation X58 I/O Hub PCI Express Root Port 1 (rev 13)
00:03.0 PCI bridge: Intel Corporation X58 I/O Hub PCI Express Root Port 3 (rev 13)
00:07.0 PCI bridge: Intel Corporation X58 I/O Hub PCI Express Root Port 7 (rev 13)
00:09.0 PCI bridge: Intel Corporation X58 I/O Hub PCI Express Root Port 9 (rev 13)
00:0a.0 PCI bridge: Intel Corporation X58 I/O Hub PCI Express Root Port 10 (rev 13)
00:14.0 PIC: Intel Corporation X58 I/O Hub System Management Registers (rev 13)
00:14.1 PIC: Intel Corporation X58 I/O Hub GPIO and Scratch Pad Registers (rev 13)
00:14.2 PIC: Intel Corporation X58 I/O Hub Control Status and RAS Registers (rev 13)
00:1a.0 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB UHCI Controller #4
00:1a.7 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB2 EHCI Controller #2
00:1c.0 PCI bridge: Intel Corporation 82801JI (ICH10 Family) PCI Express Port 1
00:1c.4 PCI bridge: Intel Corporation 82801JI (ICH10 Family) PCI Express Port 5
00:1d.0 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB UHCI Controller #1
00:1d.1 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB UHCI Controller #2
00:1d.2 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB UHCI Controller #3
00:1d.7 USB Controller: Intel Corporation 82801JI (ICH10 Family) USB2 EHCI Controller #1
00:1e.0 PCI bridge: Intel Corporation 82801 PCI Bridge (rev 90)
00:1f.0 ISA bridge: Intel Corporation 82801JIR (ICH10R) LPC Interface Controller
00:1f.2 SATA controller: Intel Corporation 82801JI (ICH10 Family) SATA AHCI Controller
02:00.0 VGA compatible controller: Matrox Graphics, Inc. MGA G200e [Pilot] ServerEngines (SEP1) (rev 02)
05:00.0 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
05:00.1 Ethernet controller: Intel Corporation 82576 Gigabit Network Connection (rev 01)
07:00.0 RAID bus controller: Hewlett-Packard Company Smart Array G6 controllers (rev 01)
root@ubuntu:/sys/bus/pci/devices/0000:00:01.0#
```

图 9-2

每一行的开头有 3 个数，第一个数是总线编号，这里都是 00（一个系统可以有 多条总线，每条总线都是一个单独的域，叫作 PCI 域）；第二个数是设备编号，可以唯一地定位一个设备；第三个数是功能编号，一个设备可以有多个功能。sys 目录下也有，建议多使用 sys 目录。

## 13. PCI 总线协议

总的来说 PCI 是一种传统的总线结构，PCI-E 则变成了目前高速设备正在普遍



采用的网络结构。通常总线结构相当于计算机网络的总线结构，所有设备共享总线，通过总线调度为各个设备提供服务。而网络结构相当于计算机网络的星型结构，系统中各个设备直接连接到交换机，交换机再连接到路由器，各个路由器之间通过路由进行转发通信。现代计算机网络思想已经全面进入硬件内部领域。

由于 PCI-E 是星型网络。每个小型的 PCI-E 系统中只有一个 RC (Root Complex, 直接连接 CPU 的设备，相当于路由器)，实际的架构需要交换机设备，该设备叫作 Switch。

每个 Switch 可以有多个口，Switch 之间可以级联。所以必然有一个 Switch 直接连接到 RC，然后向下扩展网络，每个 Switch 的出口都可以接入设备，并且只能接入一个设备。这在交换机网络中是司空见惯的事情，但是在硬件中这却是一个不小的进步。因为之前的做法相当于每个 Switch 口出来都可以是一条总线挂载多个设备。

星型的网络结构显著增加了吞吐量，提高了链路利用效率和可扩展性，USB 总线等现代总线都已经采用这种结构。USB 总线流行的根本原因可能也是因为这个架构的使用。

#### 14. 物理层

PCI-E 的物理层抛弃了 PCI 的单端信号，改用了抗干扰能力极强的差分信号。采用的编码是 8/10 编码，8/10 编码是 IBM 已经过期的专利编码，这个编码专门用于高速串行总线。由于高速串行总线要求电流总体为 0（直流平衡），所以数据流中的 1 和 0 的数目必须一样多，能够让 1 和 0 一样多的编码就是 8/10 编码。

如果连续出现 1 或 0 也会导致物理链路出现问题（耦合电容充满），所以 8/10 编码还保证连续的 1 或 0 不超过 5 个（这是标准的，PCI-E 使用的 8/10 编码可以保证 10 个位中最多有 6 个 1 或 0，从而尽量保证不连续）。编码的时候将 8 个字节分为 3 个和 5 个，然后编码为 4 个和 6 个。但是尽量保证不连续的意思是并不能一定保证，所以 PCI-E 还有额外的机制来防止连续的情况发生，这个机制叫作 CRD。

#### 15. 协议数据包

既然 PCI-E 采用了交换式架构，这也决定了其使用可交换的数据包进行通信。协议分为两层：数据链路层和事务层。链路层有链路训练的功能，总线事务也有很多种，例如存储器读写、配置读写、Message 总线事务、原子操作等。还有一些高

级功能，例如流量控制、虚通路管理等。两层都有数据报文和控制报文两种。下层为上层提供服务，上层使用下层的服务接口。

## 16. PCI 总线与 USB 总线的关系

如果使用 `lspci` 命令列出所有的 PCI 设备，就会发现，在 Linux 内核中，大部分情况下 `Usbcontroller` 是 PCI-E 的一个设备，也就是说 USB 总线是挂载在 PCI(PCI-E) 总线上的。因此到 USB 的存储数据的真实流向应该是：用户→文件系统→通用块层→SCSI→USB→PCI。大家可能会疑惑从 CPU 出来的数据明明是先经过 PCI-E，然后经过 USB Controller 到达 USB 设备，为什么内核中的数据走向是先经过 USB 设备的呢？

一个更完整的单机流程为：用户→文件系统→通用块层→SCSI 驱动→USB 驱动→PCI 驱动→PCI 硬件→USB 硬件→USB 设备。这里还是要强调一点，驱动和硬件不是一个实体。

## 9.2 USB

### 9.2.1 USB 概览

USB (Universal Serial Bus) 是一种传输协议，并不是一种数据协议，也没有任何语义上的指令意义（有一些 USB 协议的管理指令）。USB 传输协议所传输的，例如 SCSI 命令才是各个存储设备所能理解的命令，USB 的责任就是将这些命令送达并且返回命令所要求的数据。所以 USB 传输协议是不认识 SCSI 指令的，它的任务只是将上层的任何数据以 USB 的传输方式送达。

USB 作为一种传输协议，主要有 3 个优点，即集成电源、造价便宜、支持广泛。这里要说明的是，论速度，USB 不算是最快的；论价格，USB 不算是最便宜的。但 USB 软件支持系统的复杂性，在所有的传输协议里是首屈一指的。但是 USB 依然被广泛应用，以致成为全世界事实上的数据传输标准，就连英特尔和苹果共同推广的速度极高的 Thunderbird 传输协议也无法撼动 USB 的地位，主要原因有两个：集成电源和支持广泛。集成电源让其不仅可以作为数据接口，也可以作为充电接口存在，在移动设备的充电方式上，USB 口已经成为了事实上的标准。而且 USB 有广泛的大

商家支持，所以才发展得如此顺利。Intel、IBM、Microsoft、Compaq 等具有影响力的大型公司就是 USB 的创始者。

USB 是金字塔型的，与 SCSI 一样，最上层是总线的硬件控制器芯片：USB Host。根 Host 下必须挂一个 Hub。USB Hub 的存在让 USB 系统组成一颗树，可以自由扩展。USB 分为 1.0、1.1、2.0、3.0 和 3.1 几个版本，最近还推出了速度更快的 3.2 标准，传输速度和功能复杂性都在不断改变。

## 9.2.2 USB 子系统上层（USB 设备驱动层）

USB 子系统的上层就是实际的驱动程序，是要注册到系统的驱动程序列表的结构体。对 Storage 来说，在 drivers/usb/storage/usb.c 中有完整的模块初始化和卸载函数。

### 1. USB 与 SCSI 的对接

实际上 USB 的每个设备都是 SCSI 的一个 SCSI Host，所以 SCSI 模块向 USB 模块传递命令时都是直接通过调用这个 scsi\_host 所规定的接口函数。最重要的是 Queuecommand 函数，这个函数将从 SCSI 传来的命令实际挂载到 USB 子系统内部的结构体上，也是 USB 模块的最上层结构体（struct us\_data）。值得注意的是，这个 Queuecommand 函数接口虽然是在 SCSI 子系统定义的，但是其具体的实现却是在 USB 子系统中。通过这一步 USB 子系统将来自 SCSI 层的命令传输到了本层。但是，这时该命令仍然没有执行。例如 ISCSI 的对应函数是 iscsi\_queuecommand (drivers/iscsi)。新的 4.5 版本的内核已经去掉了独立的 Queuecommand 函数，变成了一个更加清晰的宏的定义，代码如下。

```
Include/scsi/scsi_host.h
#define DEF_SCSI_QCMD(func_name) \
    int func_name(struct scsi_host *shost, struct scsi_cmnd *cmd) \
    { \
        unsigned long irq_flags; \
        int rc; \
        spin_lock_irqsave(shost->host_lock, irq_flags); \
        scsi_cmd_get_serial(shost, cmd); \
        rc = func_name##_lck (cmd, cmd->scsi_done); \
        spin_unlock_irqrestore(shost->host_lock, irq_flags); \
    }
```



```

        return rc;
    }
drivers/usb/storage/scsiglue.c
static DEF_SCSI_QCMD(queuecommand)

```

可以看到 DEF\_SCSI\_QCMD 宏是在 SCSI 部分定义的,但是在 USB 的上层使用的。也就是 SCSI 的下层和 USB 的上层的粘结 (glue)。rc = func\_name##\_lck (cmd, cmd->scsi\_done); 这句话说明了实际调用的函数是 queuecommand\_lck。代码如下。

```

static int queuecommand_lck(struct scsi_cmnd *srb,void (*done)(struct
scsi_cmnd *)){
    struct us_data *us = host_to_us(srb->device->host); //usb 的最上层
    结构体
    /*每个 us_data 同时只能有一个命令,如果当前 us_data 已经有命令了,queuecommand
    将返回错误。*/
    if (us->srb != NULL) {
        printk(KERN_ERR USB_STORAGE "Error in %s: us->srb = %p\n",
            __func__, us->srb);
        return SCSI_MLQUEUE_HOST_BUSY;
    }
    /* 如果 USB 设备当前断开连接,就退出函数 */
    if (test_bit(US_FLIDX_DISCONNECTING, &us->dflags)) {
        usb_stor_dbg(us, "Fail command during disconnect\n");
        srb->result = DID_NO_CONNECT << 16;
        done(srb);
        return 0;
    }
    /* 实际的入队列,并且唤醒命令准备就绪,回调函数 */
    srb->scsi_done = done;
    us->srb = srb;
    complete(&us->cmdn_ready);
    return 0;
}

```

另外,us\_data 是 USB 子系统上层调度的实体,并不代表任何的具体设备。而这个 us\_data 是如何与 scsi\_host 关联起来的呢?在 scsi\_host 结构体的最下面,有一个域叫作 unsigned long hostdata[0]。整个 us\_data 结构体就放在这里,所以可以通过

scsi\_host 直接找到其对应的 us\_data，也就是唯一的 USB 设备。代码如下。

```
struct scsi_host {
    //……此处之上省略
    unsigned long hostdata[0] __attribute__((aligned (sizeof(unsigned
long))));
};
```

如果阅读代码会发现，在 scsiglue.c 中定义 SCSI 的接口数据结构并不是直接定义的 scsi\_host，而是定义的 struct scsi\_host\_template。这是 SCSI 的结构决定的，只需要定义这个结构体，SCSI 子系统会根据这个结构体生成对应的 scsi\_host 结构体。

## 2. USB Storage 执行 SCSI 命令

这里以 USB Storage 为例讲述。USB 设备被关注最多的方面就是存储设备，USB 的存储设备在 USB 子系统中位于 drivers/usb/storage 子目录。真正执行 us\_data 中命令的是 usb-storage 内核线程。该线程可以有多个，其启动的参数就是传入一个 us\_data。该线程会做一些列检查，例如当前是否有命令，如果没有命令就退出。最主要的是检测有命令要执行时，会调用 us\_data 结构体中注册的函数 proto\_handler 执行。

可以看出 Linux 内核以数据结构为核心的设计思想。所有的操作和操作所需要的数据都在数据结构中，但是什么时候调用这些操作，调用操作的结果怎么存储到数据结构中，则是通过一些外部的函数或线程进行的。所有的代码都围绕着数据结构为其“打工”，周边代码存在的目的是让数据结构“动起来”。

那 proto\_handler 究竟调用的什么呢？USB 子系统的存储部分根据 SC(subclass) 类型的不同定义了不同的 proto\_handler，如下所示。

```
#define US_SC_RBC          0x01          /* 闪存设备*/
#define US_SC_8020         0x02          /* CD-ROM */
#define US_SC_QIC          0x03          /* QIC-157 磁带 */
#define US_SC_UFI          0x04          /* 软盘 */
#define US_SC_8070         0x05          /* 可删除媒体*/
#define US_SC_SCSI         0x06          /* SCSI 透传*/
#define US_SC_LOCKABLE     0x07          /* 密码保护设备*/

#define US_SC_ISD200       0xf0          /* ISD200 ATA */
```

```
#define US_SC_CYP_ATACB    0xf1        /* Cypress ATACB */
#define US_SC_DEVICE       0xff        /* 设备自定义 */
```

实际上，虽然分了这么多类，但是处理函数只有两种可能。最终实际调用的函数 `us_data` 结构体的注册函数是 `transport`。

`transport` 函数根据协议不同还有两个函数（有 3 种 USB 协议），如下所示。

```
#define US_PR_CBI          0x00        /* 控制/批量/中断 */
#define US_PR_CB           0x01        /* 控制/批量 */
#define US_PR_BULK         0x50        /* 批量*/
```

但是原理都是一致的，生成并填充一个 URB，然后提交。URB 是 `usb core`（USB 总线驱动）中的内容。对于磁盘等存储设备，对应的是 `US_PR_BULK` 模式。`usb` 模块会向设备发送 3 种数据包：CBW（CommandBlock Wrapper）、CSW（Command Status Wrapper）和数据。无论如何，USB 子系统都会首先发送 CBW，只有在有数据的时候才会发送数据体，再发送 CSW 获得设备的命令执行情况，最后根据 CSW 返回的设备情况向上报告当前命令的执行是否成功。可以看出这是一个损耗很高的过程，所以当发送数据时应尽量发送多的数据。实际的发送代码位于 `drivers/usb/storage/transport.c` 中。

可以容易地想到，可以修改协议让 `usb` 模块发送多次 CBW 而只获得一次 CSW。从而理论上就可以大幅度地提高发送速度，但是也不能突破硬件上限。

### 3. USB Storage 设备的发现过程

当这个驱动扫描函数（`storage_probe`）被调用时，就会进行扫描发现过程。

`storage_probe` 包括 `usb_stor_probe1` 和 `usb_stor_probe2` 两个函数阶段，完成对 `scsi_host` 为 USB Storage 的 `us_data` 初始化时。在 `usb_stor_probe2` 末尾时还会启动另外一个内核线程 `usb-stor-scan`。这个内核线程会实际调用 SCSI 的扫描接口，填充 `scsi_host` 结构体的其他域。这里使用线程是延迟的一种手段，不让内核在这里阻塞，对 SCSI 部分内容的填充可以后续完成。

## 9.2.3 USB 子系统的中层（USB core）和下层

对于理解整个系统的架构来说，上层是最重要的，因为它展示了如何与其他的



子系统整合合作。USB 中层是 USB 子系统的核心，核心部分承上启下，定义最完整的接口和协议实现。下层则偏于硬件，一般归属于驱动类别。USB 是以 hub 为组织核心的网络拓扑，所以下层接口的核心就是 hub 的实现。无论这个设备是存储设备还是打印机等设备，最先经过的都是 core/hub.c（这里属于中层与下层的交互部分，仍归属于中层）。目前内核版本（4.10）使用 usb\_hub\_wq 的工作队列线程来执行这个硬件发现任务（例如通过扫描发现端口）。

老版本的内核 hub.c 的入口函数是 hub\_thread 线程函数，该函数循环调用 hub\_events 函数处理 hub 事件。hub\_events 中可以处理很多事件，与设备识别过程相关，且最重要的是 hub\_port\_connect\_change 函数，用于处理端口的逻辑连接或者物理连接发生变化的情况。这个函数在最新的内核版本中仍然与之前一样，代码如下。

```
static struct usb_driver hub_driver = {
    .name = "hub",
    .probe = hub_probe,
    .disconnect = hub_disconnect,
    .suspend = hub_suspend,
    .resume = hub_resume,
    .reset_resume = hub_reset_resume,
    .pre_reset = hub_pre_reset,
    .post_reset = hub_post_reset,
    .unlocked_ioctl = hub_ioctl,
    .id_table = hub_id_table,
    .supports_autosuspend = 1,
};
```

以上结构体就是一个 hub 的驱动，而 struct usb\_driver 自然就是 core 中层提供的基础组件的抽象，硬件设备都是通过这种驱动接口接入中层框架的，如图 9-3 所示。

我们看到在 hub 的驱动中，当一个新的设备接入时，会调用 hub\_probe 函数，最后还是调用到最重要的端口连接状态改变的处理函数。

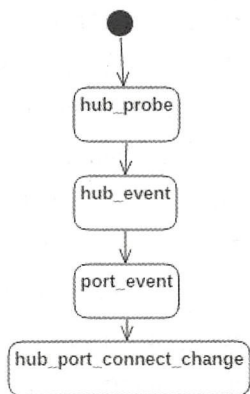


图 9-3

## 9.2.4 Platform 总线

PCI 总线只是一种 USB 挂载的总线选择。USB 总线虽然是慢速总线，需要挂载在较快的总线上作为缓存。但也有例外的情况，例如在 CPU 中直接集成 USB 控制模块，这在很多系统中是很常见的。当 USB 直接连接到芯片，或者连接到其他总线时，Linux 认为所有非 PCI 总线的设备都位于 Platform 总线上。这个总线是 Linux 虚拟的，用于统一管理。比如 Docker 也会创建自己的 Platform 总线，使用 struct platform\_driver 结构体，实现结构体定义方法的内核模块就可以实际定义一个 Platform 总线。

对于 Linux 来说，总线是这样一种设备：允许设备（逻辑设备结构体）连接到该总线，允许驱动挂载到该总线，通过总线提供的遍历方法遍历所有的设备，能够动态检测设备与总线之间的连接。所以，这就在逻辑上提供了虚拟总线的可行性。

## 9.3 用户空间的设备管理

用户空间所能见到的所有设备都放在 /dev 目录下，文件系统所在的分区被当成一个单独的设备也放在该目录下。以前的 2.4 版本的内核曾经出现过 devfs，这个文件系统实现设备管理的思路非常好，在内核态实现对磁盘设备的动态管理。可以做

到当用户访问一个设备时，devfs 驱动才会去加载该设备的驱动，甚至每个节点的设备号都是动态获得的。但是该机制的开发者不再维护它的代码，Linux 成员经过讨论，使用用户态的 udev 代替内核态的 devfs，所以现在的 devfs 已经被废弃了，现在仍然使用增强版的 devtmpfs。用户态的 udev 程序在设备发现流程的时候加载设备驱动，动态的在 /dev 目录下创建节点。/dev 只是一个目录，而不挂载 devfs 文件系统。当然，这个 udev 只是一个应用程序，还可以用别的程序替代，例如 busybox 就实现了 mdev 程序（一个简化版的 udev）完成的同样的工作。

### 9.3.1 设备变化通知用户端

系统在启动时会对设备做检测，系统启动后设备的变化系统也应该能够识别。本质上，启动时发现的硬件也是一种设备的变化。这种设备的变化仅仅内核知道是没有意义的，因为使用设备的用户是用户空间的程序，内核只是管理者，但只能管理却不能使用，资源就是无意义的存在。那么内核如何将设备的变动信息通知到用户程序呢？这就不得不说一下 uevent 机制了。

内核通过向用户空间发送 uevent 事件来通知用户空间程序设备资源的变化，事件所传递的变化的具体内容是通过 uevent 事件所附带的参数缓存来实现的。而用户空间对该事件进行响应的程序就叫作 udevd，但是这种内核通知，用户响应的机制就叫作 udev。

但这只是目前的机制。Linux 是一个不断演化的系统，之前为了完成相同的功能还使用过 hotplug 程序（有设备变动就执行一遍该程序，可重入的多次执行，/proc/sys/kernel/hotplug 定义 hotplug 响应程序），还有 devfs（提前在 /dev 目录创建了一堆节点文件，无动态性），而 udevd 是一个后台服务程序，不是像 hotplug 那样来一个信息就执行一个程序副本（所以需要考虑并发问题）。udev 这种处理消息的能力免除了可重入问题，加快了用户端响应内核设备变动的效率。

Linux 用户端要使用的每个设备都要在 /dev 目录中引用，除非更上层的封装（如 mount），然而这也离不开 /dev 文件系统中对设备的引用。所以对用户端应用来说，/dev 目录是它们与内核设备直接打交道的唯一途径。例如 drm 设备可以直接访问显卡；tty 设备可以直接访问串口；sr0 可以直接访问 cdrom；sda 可以直接访问磁盘。内核确认规定了如何使用各个设备，但是使用何种设备需要用户来指定（例如你修



改一个文件，所有操作都是在内核中驱动的，但是得先把文件所在分区分别 mount 到文件系统中)，如图 9-4 所示。

```
root@ubuntu:~/sdc1/linux-4.10.3/drivers/usb# echo 123 >/dev/pts/10
root@ubuntu:~/sdc1/linux-4.10.3/drivers/usb# who
archerbroler pts/0      2017-03-14 20:26 (172.26.80.16)
archerbroler pts/9      2017-03-14 20:27 (172.26.80.16)
archerbroler pts/10     2017-03-14 20:27 (172.26.80.16)
archerbroler pts/11     2017-03-14 23:36 (172.26.80.16)
archerbroler pts/12     2017-03-14 23:56 (172.26.80.16)
archerbroler pts/13     2017-03-15 00:44 (172.26.80.16)
root@ubuntu:~/sdc1/linux-4.10.3/drivers/usb#
```

图 9-4

如图 9-4 所示向 pts 中写入内容，以这个 pts 登录的用户终端就会收到通知，如图 9-5 所示。

```
root@ubuntu:~# 123
```

图 9-5

所以，udev 最重要的功能就是创建 dev 下的设备节点。但并不是所有做这个事情的应用都使用 udevd，这只是 udev 协议的一种广泛使用的实现软件，还有 busybox 使用的 mdev，也可以完成相同的功能。

另外，近代的操作系统倾向于把所有的服务程序纳入统一的管理中，有的管理方式是在需要时启动该程序实体，例如 inetd 服务，有的则直接整合了程序本身进管理程序，例如目前开始被广泛使用的 systemd 程序。如果你打开进程，就会发现在后台运行的可能不是 udevd 程序，而是变成了 /lib/systemd/systemd-udevd - daemon 服务。这就是被 systemd 统一管理的结果。甚至在 initrd 中也直接使用了这种服务。

那么，Linux 是使用何种通信手段与用户端的服务程序通信的呢？答案仍然是 Netlink。是不是只有 udevd 通过监听 Netlink 事件才能得到内核事件的变化呢？答案是“肯定不是”。内核在实现过程中考虑了各种情况，甚至还可以像以前一样指定 hotplug 程序。但是内核在实现 KObject 机制的同时也顺便实现了一种功能，叫作 uevent\_helper。在用户空间是 /sys/kernel/uevent\_helper。通过向这个文件写入一个程序路径，Linux 的 uevent 就会顺便通知这个程序。这个目录的存在需要内核支持，内核配置中的 CONFIG\_UEVENT\_HELPER=y 和 CONFIG\_UEVENT\_HELPER\_PATH="" 可以控制该机制。

### 9.3.2 设备类型

内核中定义的设备类型常见的有两种，即字符类和块类。这些/dev 目录下的设备并不一定都对应着具体的硬件（如 zero、tty），有的硬件可能对应着多个节点（如 sda、sda1）。大部分发挥特殊功能的设备都是字符设备，正是由于设备是可以虚拟的，所以诞生了框架设备这种新的设备子类型。

input 设备是一种字符设备，很多与输入相关的设备都使用这个 input 设备进行管理。也就是说 input 虚拟设备是为其他输入设备服务的。

与磁盘相关的设备名字一般是 sda、sdb 等，这里的 s 代表 SCSI 设备。以前还经常出现 hd、fd 等。fd 表示软盘，hd 表示 IDE 硬盘。由于 SATA 和 SCSI 已经在很大程度上合并，在软件上已经可以处理相同的命令了，所以对于只关心软件的 Linux 来说 SATA 设备也是 sd 设备。sr 表示 CD-ROM，一般还有一个 cdrom 节点文件。

tty 是串口，一般会模拟很多实例出来，通过“Ctrl+Alt+F1 (F2~F7)”组合键分别调用。还有一种在图形界面模拟串口的方式是 pty (pseudo-tty)。在 Ubuntu 的程序打开一个 terminal 就是一个 pty。

loop 文件是回环设备，也是块设备。其本身不是设备，使用命令将一个文件挂载到一个目录，这个文件就被认为是一个虚拟的磁盘，里面有分区结构，在设备中就是一个回环设备。

tty 这一整套的系统从很早的历史就开始使用了，最早的 Linux 控制终端使用两条线 (RX/TX) 来传输命令。但是这个传输的标准，例如波特率、控制字符、编码等在各个标准上都是有区别的，这就诞生了不同的驱动，当你在终端上按下“Ctrl+c”组合键后的反应就是后台驱动反映的，这些不同的驱动模型叫作 line discipline，而这种使用 RX/TX 线路组成的串口设备统一被叫作 tty 设备，更改一个 tty 终端后端采用的 line discipline 的命令是 ldattach。这些不同的 line discipline 对应同样的/dev 目录下的 tty 设备。从这里就能看出 tty 的驱动必然存在通用的部分，然后是各个 line discipline 各自的驱动。stty 命令直接操作 tty 设备的时候就可以显示和改变这些 line discipline 参数。一个 tty 设备只能由一个进程打开。也就是说，一般意义上一个 tty 上只能运行一个进程，这个进程就是 shell。

这是最原始的样子，随着多任务需求的增加。人们希望让一个 tty 上可以运行多个进程，虽然我们实际都是使用一个进程来运行 shell，但是我们希望脱离这个 tty

运行一个后台的进程。首先诞生的是前端任务和后端任务的概念。当我们在 shell 上执行一个命令时，在命令执行完毕之前，可以按“Ctrl+z”组合键让命令暂停，这是由于内核的信号 SIGSTOP (19) 可以暂停一个进程，而 SIGCONT (18) 信号可以继续这个进程的执行，并可以使用 kill -19 pid 命令来重复这个过程。shell 通过让当前进程挂起到后台暂停，从而使得交互式的 shell 又处于可用的状态，而且不必关闭之前正在运行的进程。

典型的使用场景是我们在使用 vim 编辑代码时，按下“Ctrl+z”组合键到命令行输入一些命令，然后再输入 fg 命令，将 vim 的代码调出来继续编辑。但是这种方式远远不能满足人们的需求，有一大批需求是希望进程在后台运行的，而当前 shell 依然可用。这个时候 shell 发明了“&”命令，例如我们使用 ./test & 命令就可以让程序在后台继续执行，而不是挂起到后端任务。例如 nc 命令，笔者在本机的 21 号 pts（后面会介绍 pts 的后台原理）监听 1234 端口，而在另外一个 pts 上 telnet 上来就能发现，0、1、2 号 fd（分别代表标准输入、标准输出、标准错误输出）分别都是指向了我们的 pts 的设备，所以当前的 shell 仍然会收到这个命令的执行结果的输出。如图 9-6 和图 9-7 所示。

```
root@ubuntu:~# nc -l 1234
[3] 26777
root@ubuntu:~# ps aux|grep "nc -l"
root    26777  0.0  0.0  9184  888 pts/21    S   23:28   0:00 nc -l 1234
root    26779  0.0  0.0 14224 1012 pts/21    S+  23:29   0:00 grep --color=auto nc -l
root@ubuntu:~# ll /proc/26777/fd
total 0
dr-x----- 2 root root 0 Mar 30 23:29 /
dr-xr-xr-x 9 root root 0 Mar 30 23:29 /
lrwx----- 1 root root 64 Mar 30 23:29 0 -> /dev/pts/21
lrwx----- 1 root root 64 Mar 30 23:29 1 -> /dev/pts/21
lrwx----- 1 root root 64 Mar 30 23:29 2 -> /dev/pts/21
lrwx----- 1 root root 64 Mar 30 23:29 3 -> socket:[483565]
root@ubuntu:~# 1234
```

图 9-6

```
root@ubuntu:~# telnet 127.0.0.1 1234
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^\''.
1234
```

图 9-7

事实上，我们使用 fork 和 signal 能够完成更多的控制。例如在后台执行一个进程使用 fork 技术，在 shell 当前的界面阻塞执行直接使用 execve 就可以了，不必产生新的进程。由于在一个 shell 结束的时候，它会给所有由这个 shell 产生的进程发送 SIGHUP (1) 信号，导致后台程序的退出，nohup 命令则会截断这个信号，让 shell



启动的进程收不到这个信号，进程就不会退出。或者是显示在进程中调用 `signal(SIGHUP, SIG_IGN)` 来忽略这个信号。

后来随着网络的发展，人们开始远程访问 `tty`，例如 `openssl`。这就诞生了虚拟 `tty` 的需求，这就是 `pts`。但虚拟的 `pts` 会仍旧使用 `tty` 的串口驱动，后端仍旧有原来 `tty` 拥有的各种类型的标准，只是在底层的读写设备上一个是串口线路，另一个是虚拟文件。与物理的 `tty` 不同的是，每一个 `pts` 都是按需创建的。你会发现 `/dev/pts/` 下面的文件只有在有使用者的时候才会存在。这个设备在内核中位于对应的设备驱动的结构体中。里面存储了各种终端相关的信息。而创造这些 `pts` 设备的方法是通过 `/dev/ptmx` 文件。代码如下。

```
open('/dev/ptmx',...)
pts = open('/dev/pts/3',...);
dup2(pts, 0); // 对应 stdin
dup2(pts, 1); // 对应 stdout
dup2(pts, 2); // 对应 stderr
close(pts);
execl("/system/bin/sh", "/system/bin/sh", NULL);
```

通过 `ptmx` 就复制出了 `pts`，也就是说 `ptmx` 实际上是 `pts` 的原型。典型的是 `sshd`，在一个用户通过 SSH 连接上主机的时候，它会屏蔽掉 `SIGWINCH` 信号，以防止 `pts` 设备里面存储的屏幕大小的真实展示，而使用 `sshd` 自己的配置进行设置。除此之外，还可以通过 `openpty`、`forkpty` 函数进行 `pty` 的创建。

介绍几个相对高级的命令。`inputattach` 命令可以为底层添加一个虚拟的串行硬件。我们这里用它来为底层添加一个虚拟的键盘，这个键盘的输入来自我们的串口。`Socat` 命令是一个非常强大的命令，它可以用来连接两个文件（Linux 下的一切皆文件概念）的输入、输出。我们这里用它来连接一个 `pts` 设备和一个命名管道文件。这个命名管道文件就作为 `pts` 设备的输入缓存，如图 9-8 所示。代码如下。

```
root@ubuntu:~/tty# socat pty,link=my_pty pipe:my_pipe&
[7] 29484
root@ubuntu:~/tty# inputattach --daemon -ps2ser my_pty
root@ubuntu:~/tty# echo
12345567489345t4jklfdwshfjkaahdsjkhqwklfmnewklgjklbgkldfmlm8678 >my_pipe
root@ubuntu:~/tty# ll
total 8
```

```
drwxr-xr-x  2 root          root          4096 Mar 31 02:00 ./
drwxr-xr-x 25 archerbroler archerbroler 4096 Mar 31 02:04 ../
prw-r--r--  1 root          root           0 Mar 31 02:02 my_pipe|
lrwxrwxrwx  1 root          root          10 Mar 31 02:00 my_pty ->
/dev/pts/2
root@ubuntu:~/tty#
```

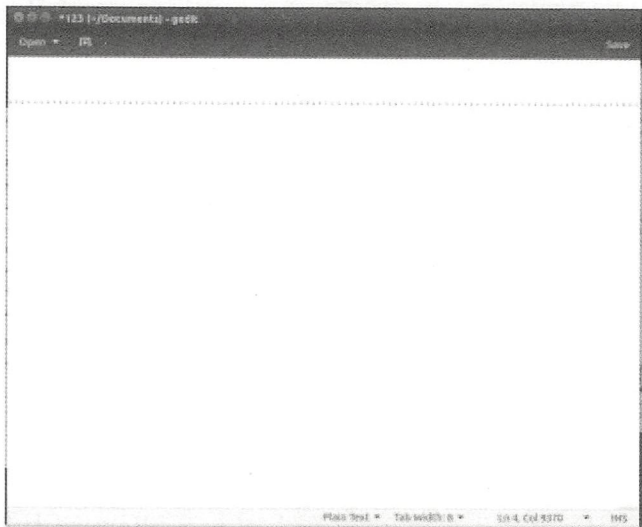


图 9-8

可以看到，我们用 `socat pty,link=my_pty pipe:my_pipe&` 命令创建了一个 `my_pty` 设备，这个设备被设置为 `pts/2` 的软链接（由 `socat` 新建），同时还创建了一个 `my_pipe` 管道文件（也就是 FIFO 文件）。然后 `socat` 将这两个文件的输入、输出连接起来了。这样向管道文件里写入内容就相当于写入到了 `pts/2` 设备中。`inputattach --daemon -ps2ser my_pty` 命令又将 `pts/2` 这个串口模拟成了一个底层的键盘输入。这里不用关心键盘标准的编码，而应关心这套连接流程。所以随机地向 `my_pipe` 文件中写入内容，在图形界面打开一个文本文件，就会发现发生了实际的键盘输入行为。从 `pts` 的架构上可以理解 Linux 的驱动架构设计思路，同时 `inputattach` 命令的使用又重复印证了设备极其容易虚拟的特性。

有一个更有趣的例子。`slip` 是一种可以在网络上传输的串口协议，属于 `line discipline` 的一种。下面实际使用这种协议观察效果，即使用一台机器的两个 `pts`，代码如下。

Pts1:

```
root@ubuntu:~/tty# socat tcp-listen:1234,reuseaddr pty,link=my_slip&
[7] 30962
```

Pts2:

```
root@ubuntu:~/slipb# socat -v -x pty,link=my_slipB tcp:127.0.0.1:1234&
root@ubuntu:~/slipb# ldattach SLIP my_slipB
```

```
root@ubuntu:~/slipb# ifconfig sl1 192.168.1.2 pointopoint 192.168.1.1 up
```

```
root@ubuntu:~/slipb# ifconfig
```

```
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:65536 Metric:1
            RX packets:4176014 errors:0 dropped:0 overruns:0 frame:0
            TX packets:4176014 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1
            RX bytes:363748335 (363.7 MB)  TX bytes:363748335 (363.7 MB)
```

```
sl1         Link encap:Adaptive Serial Line IP
            inet addr:192.168.1.2  P-t-P:192.168.1.1
```

```
Mask:255.255.255.255
```

```
            UP POINTOPOINT RUNNING NOARP MULTICAST MTU:296 Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:10
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Pts1:

```
root@ubuntu:~/tty# ldattach SLIP my_slip
```

```
root@ubuntu:~/tty# ifconfig sl0 192.168.1.1 pointopoint 192.168.1.2 up
```

```
root@ubuntu:~/tty# ifconfig
```

```
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING MTU:65536 Metric:1
            RX packets:4173513 errors:0 dropped:0 overruns:0 frame:0
            TX packets:4173513 errors:0 dropped:0 overruns:0 carrier:0
```



```

collisions:0 txqueuelen:1
RX bytes:363534760 (363.5 MB) TX bytes:363534760 (363.5 MB)

sl0      Link encap:Adaptive Serial Line IP
         inet addr:192.168.1.1 P-t-P:192.168.1.2
Mask:255.255.255.255
         UP POINTOPOINT RUNNING NOARP MULTICAST MTU:296 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:10
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sl1      Link encap:Adaptive Serial Line IP
         inet addr:192.168.1.2 P-t-P:192.168.1.1
Mask:255.255.255.255
         UP POINTOPOINT RUNNING NOARP MULTICAST MTU:296 Metric:1
         RX packets:0 errors:0 dropped:0 overruns:0 frame:0
         TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:10
         RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Pts2:
root@ubuntu:~/slipb# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.041 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.091 ms
^C
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.041/0.065/0.091/0.020 ms

Pts1:
root@ubuntu:~/tty# ps aux|grep socat
root      30962  0.0  0.1  30664  3252 pts/21    S      02:43   0:00 socat
tcp-listen:1234,reuseaddr pty,link=my_slip
root      30963  0.0  0.1  30664  3160 pts/19    S      02:43   0:00 socat -v
-x pty,link=my_slipB tcp:127.0.0.1:1234
root      31037  0.0  0.0  14224   972 pts/21    S+     02:48   0:00 grep

```

```

--color=auto socat
root@ubuntu:~/tty# kill 30962

Pts2:
root@ubuntu:~/slipb# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
From 124.250.249.246 icmp_seq=1 Destination Net Unreachable
From 124.250.249.246 icmp_seq=2 Destination Net Unreachable
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 0 received, +2 errors, 100% packet loss, time 1001ms

```

可以看到，当 socat 启动时，我们设置的 slip 链路是通的。但是当 socat 命令关闭时，pts2 就 ping 不通了。证明通过上述流程我们建立了一条 slip 链路。建立 slip 链路除了使用 socat 这种通用的命令外，还可以使用专用的 slattach 命令。

### 9.3.3 内核数据结构的面向用户组织 KObject

Linux 内核用的一种组织数据的方式是实现一个结构体(或一种数据组织方式)，为这个组织方式的每个元素定义了结构体。任何其他的部分想要使用这个数据组织方式，只需要包含对应的结构体就可以把自己安放在数据结构的特定位置。例如 list 数据结构的实现，还有一个很重要的数据结构就是 KObject。

KObject 也是内核预先设计好的数据结构组织方式，是一个树形的结构的实例。每一个 KObject 都是这棵树的一个节点，每一个 KSet 都是这棵树的一个非叶子节点，里面包含了 KSet 或者一些 KObject。按照定义，一个 KSet 也是一个 KObject。包含 KSet 的可以是 KSet，还可以是该组织方式定义的最高层的分层数据类型 KSubsystem。从树的角度看，KSubsystem 与 KSet 没有区别，但是这个数据组织方式是在树的基础上定义的，对最上层的树做了额外的区别于 KSet 的定义，就命名为 KSubsystem。

# 10

## 第 10 章

### 二进制

ELF (Executable and Linkable Format) 的名字清楚地表达了这种文件格式的目的是用来执行和链接的。链接是一个很重要的概念，在 GCC 的世界里它既包括编译期的链接，也包括运行期的动态链接，但是在 Golang 的世界里它只包括编译器的链接。

ELF 是 Linux 下组织二进制的格式，除此格式之外，二进制话题还包括兼容性问题 and 函数调用约定 (ABI)。ELF 格式本身在编译期和执行期都有重大的作用，几乎所有的二进制的安全性都是由 ELF 格式来提供保护的，病毒感染也是通过 ELF 格式来进行的。

Linux 上所有程序的执行，包括 shell 脚本，最终都要依赖于 ELF (比较老的系统是 a.out)，shell 脚本所执行的命令大多都是以 ELF 格式组织的二进制小程序，shell 本身 (例如 bash) 在 Linux 系统上也是以二进制的形式进行存放、组织和执行的。

#### 10.1 函数调用

##### 10.1.1 函数调用约定

目前几乎所有的编程语言都离不开函数和参数的概念，而这个概念是编程语言级别的，而不是硬件级别的，也就是说在硬件中本来没有函数的概念。只是函数用



得太普遍了，硬件开始为函数准备专用的指令。

我们以 x86 的硬件为例。CPU 的功能是计算、读取数据、执行指令，这里面的问题就是指令如何执行。我们完全可以按顺序执行所有的指令，能够达到计算机的计算目的。但是这样在使用者来看是不现实的，完全按顺序执行代码在编程的初期就被发现不适合于开发，于是人们增加了循环、判断、跳转和函数。也正是这些在业务上的判断能力和在工程上的逻辑组织能力，才使得流程化的指令能够与人类的逻辑相对应，使得大规模软件成为了可能。

函数（方法）是几乎所有编程语言的组织基础，但是仍然有不使用函数的编程方式，通过大量地使用 `label` 和 `jump`，而在不同的过程代码中跳转于高性能编程里。因为这样省去了函数调用的开销（入栈、出栈、保存上下文等）。但是随着函数的发展，纯粹的函数式编程，例如 `haskell` 也被发明出来了，其提供了优美、强大、无副作用的编程方式。

函数的典型特点是传递参数、返回结果。几乎所有的编程语言都需要设计如何传递参数，如何返回函数执行的结果，在 C/C++ 的世界里，通常都可以传递多个参数、返回一个结果。x86 的 CPU 在寄存器上只提供了一个寄存器作为返回值的存储位置，但是这并不是说所有的语言都必须只能返回一个结果。

芯片只是规定了指令集，指令集中的指令都是可以执行的正确指令，而函数是语义级别的功能块，如何让函数的“大厦”在指令集之上建立起来需要靠函数调用约定。函数调用约定主要解决以下 4 个问题。

（1）参数以什么顺序入栈或者以什么顺序进入寄存器完成传递？

（2）调用其他函数的时候要保存本函数的寄存器现场，谁来保存？保存哪些寄存器？

（3）函数退出时要恢复调用者的寄存器现场，是调用者恢复还是被调用者恢复？恢复哪些寄存器？

（4）如何给函数命名？这里的命名是指如何编码参数和返回值类型到函数名中。一般编译之后的代码的函数名都不是代码中编程语言规定的函数名，而是根据这个生成的，进而存储到 ELF 文件的符号表里。

针对这几个问题的答案，有几种比较著名的约定：`stdcall`、`cdecl`、`fastcall`、`thiscall`、`naked call` 等。这些不同的选择都是由编译器做出来的，并不是由操作系统选择的，例如在 Linux 系统下的 GCC 和 Windows 操作系统下的 `msvc` 在编译 C 程序的时候就会采用不同的调用约定。而在 Windows 操作系统下的 `msvc` 和 `mingw` 也会采用不同

的约定。从这里我们更容易理解，调用约定是编译时的决策方式。

在 Win32 API 调用的时候，会采用 `stdcall`。这个调用约定如下。

- (1) 参数从右向左入栈。
- (2) 由函数调用者将参数入栈。
- (3) 函数执行结束的时候由被调用函数将寄存器恢复。
- (4) 函数名自动加前导的下画线，后面紧跟一个 `@` 符号，其后紧跟着参数的尺寸。

举个例子（32 位系统环境），代码如下。

```
int test(int a, int b){
    return a+b;
}
```

经过 `stdcall` 约定的编译之后，函数名就会变为 `_test@8`。而这个函数的最后一行汇编就是 `ret 8`，这就表示恢复堆栈。当这个函数被 `x` 函数调用的时候，`x` 函数需要进行入栈，代码如下。

```
push    b
push    a
call    test
```

这样就能够完成 `test` 函数的调用准备工作了。

现在计算机系统已经全面进入 64 位时代，在 64 位时代，在 Windows 操作系统下的 `msvc` 采用了 `fastcall`，而在 Linux 系统下的 `GCC` 选择了 `System V AMD64 ABI`。我们再以 `System V AMD64 ABI` 为例进行分析。代码如下。

```
//g++ -c main.cpp -std=c++11 -o main.o
#include <cstdint>
uint64_t test(uint64_t a, uint64_t b) __attribute__((noinline));
uint64_t test(uint64_t a, uint64_t b){
    return a+b;
}
int main(){
    test(1,2);
}
```

这段程序显示禁止了内联，然后进行编译，并且查看反汇编的结果，代码如下。

```

broler@ubuntu:~$ objdump -d main.o
main.o:      file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <_Z4testmm>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov     %rsp,%rbp
   4:   48 89 7d f8             mov     %rdi,-0x8(%rbp)
   8:   48 89 75 f0             mov     %rsi,-0x10(%rbp)
  c:   48 8b 55 f8             mov     -0x8(%rbp),%rdx
 10:   48 8b 45 f0             mov     -0x10(%rbp),%rax
 14:   48 01 d0                add     %rdx,%rax
 17:   5d                      pop     %rbp
 18:   c3                      retq

0000000000000019 <main>:
 19:   55                      push   %rbp
1a:   48 89 e5                mov     %rsp,%rbp
1d:   be 02 00 00 00          mov     $0x2,%esi
22:   bf 01 00 00 00          mov     $0x1,%edi
27:   e8 00 00 00 00          callq  2c <main+0x13>
2c:   b8 00 00 00 00          mov     $0x0,%eax
31:   5d                      pop     %rbp
32:   c3                      retq

```

我们从此段代码中可以看出 System V AMD64 ABI 的调用约定有如下特点。

(1) 在 main 函数的 0x1 和 0x2 两个立即数入栈的时候, 函数调用并没有使用栈, 而是直接移动到了寄存器中。事实上, System V AMD64 ABI 规定参数按照从左到右对应 RDI、RSI、RDX、RCX、R8、R9 的顺序直接放入寄存器, 寄存器不够用才会入栈。这里特殊的是 main 函数在入栈的时候使用的是 edi 和 esi 这两个寄存器。这两个寄存器实际上是 rdi 和 rsi 的 32 位部分, 因为这里的 1 和 2 立即数不需要使用 64 位的大小。

(2) main 和 test 函数在执行的开头都会将 rbp 入栈, rbp 保存上一个栈帧的地址, 当函数执行结束的时候都要将 rbp 继续 pop 出来。也就是说保存现场和恢复现场都是函数的被调用者的工作。在函数开始的时候 mov %rsp, %rbp 则是将当前的栈指针赋值给 rbp, 如此 rbp 在本函数执行期间就指向本函数的栈基地址了。



(3) test 函数被编译为 `_Z4testmm`，前面的 `_Z` 表示全局；4 表示标识符的长度。就像其他的调用约定一样，System V AMD64 ABI 也有一套完整的规定。

## 10.1.2 栈结构

x86 与 x64 架构提供了栈的寄存器指针，但是并不规定怎么使用这个栈，例如参数入栈的先后顺序、返回值放在哪里、两个调用之间是否要空点空间。栈是函数调用的核心话题，一般情况下 `rbp` 和 `rsp` 配合，一个表示本栈帧的基地址，一个表示在本栈中的当前使用地址。

在 x86 时代，常用的调用栈有 `stdcall`、`thiscall`、`fastcall`、`cdecl`，这几种在对栈的使用上有区别。在 x64 时代，主流应用只剩下 `fastcall` 和 `system v amd64 ABI`。例如 `stdcall` 的调用约定意味着：参数从右向左压入堆栈；函数自身修改堆栈；函数名自动加前导的下划线，后面紧跟一个 `@` 符号，其后紧跟着参数的尺寸，`stdcall` 因为早期用在 `pascal` 才有此殊荣。C 语言的默认是 `cdecl`，`cdecl` 调用约定的参数压栈顺序和 `stdcall` 是一样的，参数首先由右向左压入堆栈。所不同的是，函数本身不清理堆栈，调用者负责清理堆栈。由于这种变化，C 语言函数调用约定允许函数的参数的个数是不固定的，这也是 C 语言的一大特色。`thiscall` 是为了解决面向对象的函数调用要默认传输 `this` 指针，所以是 C++ 的默认调用方式，参数从右向左入栈。

而 `fastcall` 使用寄存器来传递参数，因为在 x64 环境中，寄存器有很多，所以规定了 `fastcall` 的前 4 个整数和浮点都放入寄存器中，超过的部分才放入栈中。所以使用 `fastcall` 可以显著加快调用速度。正因如此，在写代码的时候尽量使用 4 个以下的函数参数。`fastcall` 也保留了 `cdecl` 的灵活性，由调用者清理栈，所以也可以做到参数不固定。但是栈可能会有一块额外的空间，x64 会默认在站上分配一个备份空间，用来 `core dump` 分析的时候方便。这个空间保存了每次发生函数调用的寄存器的情况。如果开了编译器优化，这个空间一般就不会被保留了，这种 `fastcall` 一般用于微软的 x64 系统上。

我们这里只讨论 Linux。可以看到在 x86 体系下，C 语言函数默认是 `cdecl` 的调用方式，而在 x64 体系下，函数使用 `system v amd64 ABI`。由于未来大部分系统都是 64 位体系，所以 `system v amd64 ABI` 是我们最关心的调用规范。

例如下面这个最简单的多参数函数，代码如下，对应的 `cdecl` 栈，如图 10-1 所示。

```

void foo(long a, long b, long c, long d, long e, long f, long g, long h)
{
    long xx = a + b + c;
    long yy = d + e + f;
}

```

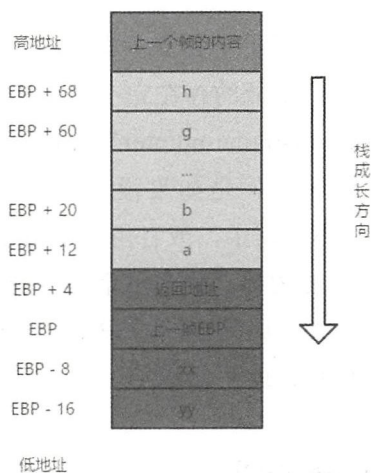


图 10-1

由于栈是从高向低生长的，在 `cdecl` 调用规范下，参数从右到左先依次入栈，然后是返回地址，然后是上一帧的 EBP，也就是上一栈帧的开始位置。然后就是具体的局部变量。上述代码对应的 AMD64 栈，如图 10-2 所示。

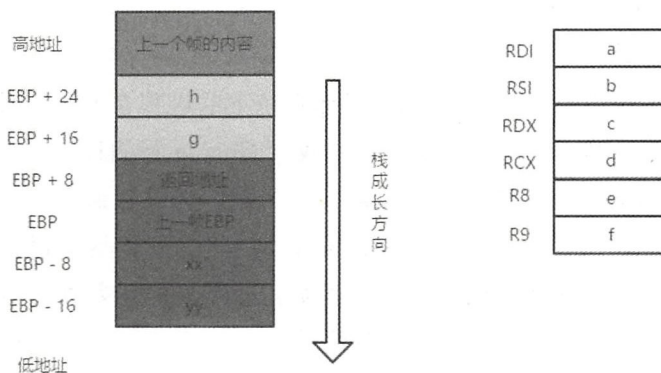


图 10-2

而在 AMD64 调用约定下，首先是前 6 个参数全部入栈，然后是从右向左入栈其他参数，接着是返回地址和上一帧的 EBP，最后是局部变量。可以看出这与 cdel 几乎一样，只是前 6 个参数入栈这一点不一样。Linux 用这种方式与 x86 保持一定程度的一致性，Windows 操作系统也有类似的 `_fast` 用于保持一定程度的前后一致性。

这里重点介绍一下 EBP (RBP)。这个域并不是必须存在的，可以看出来即使没有这个域，函数也能够被调用和返回。这个域的存在完全是为了调试用途的，所以它是可以被关闭的。使用 GCC 的 “`-fomit-frame-pointer`” 选项就能关闭这个域，这无疑会加速进程的执行，只是影响不大。但是如果没有了这个域，当程序出现 `coredump` 时，你会发现 `gdb` 的 `bt` 命令不能用了，或者用 `objdump` 反汇编出来的 `.txt` section 里面的代码没有了函数之间明显的边界。因为每个当前的栈帧都使用这个域指向上一个栈帧，如此就形成了一个链表。如果去掉 EBP 域，栈帧就无法回溯，只知道前面都是栈数据，但具体是什么栈数据就不知道了。

## 10.2 Linux 的二进制兼容性问题

Linux 上的二进制有一个显著的特点就是可移植性不强，在不同的发行版本之间，不同的内核版本之间，程序往往是不能通用的。

### 10.2.1 进程的执行

我们都知道一个现象，在 Windows 操作系统下的进程在 Linux 系统下无法用鼠标双击打开它，反之也一样。但是用 C 语言或者 Golang 写的程序在 Linux 系统下编译和在 Windows 操作系统下编译都可以执行。当然，如果你调用了操作系统特有的系统调用也是不可以执行的，确切地说是编译不通过。我们这里讨论没有调用操作系统相关的系统调用，都使用标准的 C 库函数。标准 C 库函数在后台调用的系统调用，但是这个转换工作分别在不同的操作系统的不同 C 库实现中完成。

为什么没有调用操作系统相关的系统调用还无法执行呢？简单地说是因为在 Linux 系统下编译的是 ELF 格式，在 Windows 操作系统下编译的是 exe 格式。这个



二进制格式是在内核中支持的，因为当调用了加载可执行程序的系统调用，内核必须要知道它加载的可执行文件的格式，以便从中识别信息（例如是 32 位架构还是 64 位架构；数据是大端存储还是小端存储的；符号表放在哪里；程序的入口在哪里）。这个对存储格式的识别过程有点像文件系统的运作方式，内核必须要清楚地知道不同文件系统的组织格式，才能正确地索引和修改里面的数据。让内核拥有特定格式识别能力的机制就叫作驱动。Windows 操作系统没有 ELF 驱动，Linux 内核里也没有 EXE 驱动。由于 Linux 的开源特性，完全可以写一个内核的 EXE 驱动，让 EXE 程序可以直接在 Linux 系统中执行。而 Windows 操作系统中的 Linux 子系统也能够识别 ELF 格式的文件，从而执行。在实际的汇编代码执行上，无论是 Windows 操作系统还是 Linux 系统，执行起来无任何区别，因为都是以 CPU 统一的 opcode 执行的。

但是在 Windows 操作系统中编译代码使用的基础库只能在 Windows 系统上运行，而这个基础库规定了进程做系统调用时函数参数该以何种顺序压入堆栈，该如何进行系统调用（Linux 和 Windows 陷入系统调用的方式不一样）。也就是说，如果基础库设计得足够好，能在两个操作系统之间兼容（符号表是一样的），对不同底层接口让基础库去处理也可以。且不能忘记一个程序会依赖很多动态库，这些动态库也是与系统相关的。有的代码甚至会绕过基础库，直接进行系统调用。还有进程执行需要加载器，加载器也得能够识别其他平台的格式，这也是 wine 能够工作的基础。在 Linux 系统下的 wine 程序就是通过将底层的所有不同的系统接口做转换，让 EXE 二进制在 Linux 系统上兼容。所以可以看出，如果内核的系统调用足够多且与 Windows 操作系统一致，再实现一些兼容的基础库，Linux 也是可以高效兼容 EXE 程序的。

像 Linux 和 Windows 的这种情况叫作二进制不兼容，即 ABI 不同。ABI 会规定底层的调用和参数传递，以及二进制文件布局的具体格式。如果一个内核支持一个 ABI，那么无论在什么操作系统中，一个二进制文件进行一次编译就可以处处被执行了。

## 10.2.2 同操作系统下的 ABI

在架构上，必须要区分 x86 和 x64 两种架构，一般的 x64 的机器都能运行 x86

的程序，但是如果把程序编译为 x86，就得面对大量的 x64 服务器的性能瓶颈。这个架构上的区别在任何平台上都是一样的，并不是 Linux 特有的问题。

而 ABI 的不同则是 Linux 内核和 glibc 的升级的规范变化导致的。不同的 ABI 程序和库在不同的环境下，有很高的概率是不能运行的，除非是低版本、最原始的 ABI 在现代系统上跑，一般都可以向下兼容。而这种不兼容主要发生在 C++ 身上，因为 C++ 近几年特性改变的速度相对较快，管理困难。x86 上常见的 ELF ABI 有以下 3 个。

- OS/ABI: UNIX - Linux。
- OS/ABI: UNIX - System V。
- OS/ABI: UNIX - GNU。

其中 GNU 和 Linux 是相同的，只是使用不同版本的 readelf 会现实不同的结果。而 System V 则是最古老的，也是兼容性最好的 ABI。老一些的系统只识别 System V 的 ABI。但是 system v ABI for x86\_64 却是比 Linux 还要先进的 ABI，因为 64 位系统沿用了 32 位的 system v ABI，并且进行了升级。X86\_64 这个 ABI 把大部分的参数转由寄存器传递，而不是由栈传递。在二进制安全上对栈的使用的减少，就增加了以往的缓存溢出的难度，还有 x32 上的 return to libc 等攻击的手法也得变通，提高难度。

一个 ELF 支持被编译成各种大小端，我们不能用编译成大端的 ELF 在小端的内核上执行。但是考虑到 Intel 的 CPU 全部是小端，所以在 Intel 平台上编译部署不需要过多地考虑这个问题。

众所周知，硬件平台不一样，指令集就不一样，二进制几乎没有可移植的能力。Sparc 的二进制除非用全虚拟机的方式，否则不可能在 Intel 的 CPU 上运行。

### 10.2.3 内核版本

在编译 GCC 的时候可以使用 `--enable-kernel` 指定最低支持的内核版本，这个选项会在 ELF 头部添加 `.note.ABI-tag`。如果你用 readelf 读取头部，ELF 文件头部，如图 10-3 所示。

```

archerbroler@ubuntu:~$ readelf -n /bin/ls
Displaying notes found at file offset 0x00000254 with length 0x00000020:
  Owner                Data size    Description
  GNU                  0x00000010  NT_GNU_ABI_TAG (ABI version tag)
    OS: Linux, ABI: 2.6.32

Displaying notes found at file offset 0x00000274 with length 0x00000024:
  Owner                Data size    Description
  GNU                  0x00000014  NT_GNU_BUILD_ID (unique build ID bitstring)
    Build ID: eca98eeadafddff44caf37ae3d4b227132861218

```

图 10-3

ABI tag 最低支持的内核版本是 2.6.32, 其中“-n”选项可以直接读取.note.ABIN-tag section 的内容, 以可读的方式打印出来。

运行时会直接检查这个与当前内核版本之间的区别, 不满足就会“FATAL: kernel too old”。你也可以通过 file 命令发现这种不兼容的情况, 这个命令会输出二进制的最低兼容的版本。有的库也携带着这个限制, 有的则没有, 你的进程所依赖的任何一个库满足了这个限制都会导致二进制执行不成功。查看二进制最低支持的内核版本, 如图 10-4 所示。

```

archerbroler@ubuntu:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=eca98eeadafddff44caf37ae3d4b227132861218, stripped

```

图 10-4

## 10.2.4 库

GCC 倾向于动态编译和动态加载, Golang 就是反对这一点的代表。但是动态编译携带库确实能减小大工程结果文件的大小, 而且这是 Linux 系统的传统价值观, 所以在很多地方动态加载还是很有必要的。

而各个系统所有的库的版本都不一样, 很多库的调用名的 symbol 都会在后面追加版本号, 如果版本号不匹配, 则库不能通用。如果出现了这个问题, 使用命令 `strings /lib64/libc.so.6 |grep GLIBC_`, 通过这个命令可以排查出, 大部分库的问题的根源都在 libc, 但这不是绝对的。libgcc\_s.so.1 是 GCC 的组件, 编译和运行时都需要, 一个版本的 GCC 编译的程序常常不能在装有另一个版本的 GCC 的平台上运行, 所以从高到低版本的迁移都需要带着它。libc.so.6 是最底层的库, 操作系统和其中所有应用程序几乎都依赖这个库, 是应用程序能够跟操作系统通信的基础。Libc 库有两套, 即 UNIX 中的 libc 和 GNU 开发的 glibc, 虽然名字是 libc, 但事实上就是 glibc,



两者功能没有太大差别。libm.so.6 则是对 libc 里面的数学部分优化后的版本，一般情况下也需要考虑移植问题。

下面看一个实际的 makefile 文件，代码如下。

```
BUILD=../build
INCLUDES=-I$(BUILD)/include -ICommon/ -IUpload/
DYN_LINK=-L$(BUILD)/bin/libs/ -lstdc++ -lpthread -lsqlite3 -lrt
-lnghttp2_asio -lcurl -lnghttp2 -lssl -lcrypto -ldl
STATIC_LINK=$(BUILD)/libs/libbackend.a $(BUILD)/libs/libgenerated.a
$(BUILD)/libs/libids.a $(BUILD)/libs/libcommon.a
$(BUILD)/libs/libprotobuf.a $(BUILD)/libs/libconfig.a
$(BUILD)/libs/libboost_thread.a $(BUILD)/libs/libboost_system.a
$(BUILD)/libs/libboost_filesystem.a
$(BUILD)/libs/libboost_log_setup.a $(BUILD)/libs/libboost_log.a
$(BUILD)/libs/lib
boost_program_options.a $(BUILD)/libs/libboost_regex.a
$(BUILD)/libs/libboost_filesystem.a
all:subdir
g++ -g -std=gnu++0x main.cpp $(INCLUDES) -L$(BUILD)/libs
-lconfig $(STATIC_LINK) $(DYN_LINK) -
Wl,-rpath=../libs/ -Wl,--dynamic-linker,../libs/ld-linux-x86-64.so.2
-o mybin
```

如此编译，生成的最终的二进制就是可移植的，但是需要自己携带库。“-Wl,-rpath=../libs/”选项就是指定携带的库都位于可执行文件的../libs/，而为了最大程度地可移植，甚至携带了加载器“-Wl,--dynamic-linker,../libs/ld-linux-x86-64.so.2”，所用到的动态库首先用-L选项指定执行.so文件存放的相对目录，然后使用-l选项指定名称即可（-lconfig），而所用到的静态库则直接使用绝对路径就可以编译进来，运行时并不依赖这些静态库。

另外在编译的时候会使用本机的库，可以通过 g++ -v test.cpp 命令查看结果。通常会有 5 个独立的文件需要单独链接。这 5 个文件也是可移植性的链接所要考虑的。可以直接复制它们到自己的工程目录中，然后手动添加如下静态链接路径：

```
$(BUILD)/libs/crt1.o $(BUILD)/libs/crti.o $(BUILD)/libs/crtbegin.o
$(BUILD)/libs/crtend.o $(BUILD)/libs/crtn.o
```

## 10.2.5 编译器

GCC 的 5.1 版本的编译器会在编译时做大量激进的优化，但是有的优化只对于最新的 CPU 特性有效，老一些的 CPU 在硬件层面就不支持这些优化，所以如此编译的程序就有兼容性的问题。解决编译器兼容性问题的方法是用更老的编译器。

GCC 针对符号类型添加了 UNIQUE 类型，而这个是 ELF 标准没有定义的，很早期的 Linux 版本也不会支持，所以就会出现运行错误。但是由于这个特性由于添加的较早，所以现在使用的 Linux 一般都不会遇到这个问题。

在编译高度可移植性的程序时，一种方法是用尽量低版本的编译器；另一种方法是用内核支持的任意版本的编译器，但是要携带可移植的库，并且不要使用高版本中硬件不支持的特性。

## 10.3 ELF 文件执行原理

### 10.3.1 ELF 文件分类

ELF 是一个尽可能可以通用的格式，其并不只是用来表达可执行文件的。典型的 ELF 类型有 ET\_REL、ET\_DYN、ET\_EXEC、ET\_CORE 和 ET\_NONE。这是规范定义层面的，在 Linux 系统中，具体的表现就是 ET\_REL 类型代表的是编译后的.o 文件，也就是 PIC（位置无关）代码，ET\_DYN 就是生成的.so 库文件；ET\_EXEC 就是最终的 ELF 可执行文件；ET\_CORE 就是 coredump 产生的文件；ET\_NONE 则表示未定义的格式。

所以所有作用于 ELF 文件格式的工具，例如 readelf，都可以用于以上的各种格式的二进制中，而不仅仅是可执行文件中。ELF 有两个组织上的头部，一个是 program header；另一个是 section header。由于可执行文件中可以不需要 ELF 标准的 section header 的头部，只需要 program header，而 objdump 这种工具可能是完全依赖 section header 的头部信息的，大部分的 binutils 工具都是如此，所以很多工具在通用性上可能不尽如人意，但是能够处理大部分的情况。

## 10.3.2 ELF 文件格式

磁盘存储结构一般都要有头部，ELF 文件也一样。头部有 3 部分，ELF 头部、segment 头部（program header）和 section 头部。其中一个可以正常运行的二进制文件只有一个 ELF 头部，一个 segment 头部，没有或者有一个 section 头部。一个 segment 逻辑上包含多个 section。

那么 segment 和 section 又是什么概念呢？segment 常见的有 PT\_LOAD、PT\_DYNAMIC、PT\_INTERP、PT\_NOTE、PT\_PHDR 等。下面我们来思考进程执行的必要条件。

### 1. Segment

二进制文件在磁盘中的布局并不是内存中的布局，所以需要有一个从磁盘到内存的映射和一个实现这个映射的程序，还有 Linux 上的二进制文件一般需要加载共享库（例如 libc 几乎是必备的），这个工作并不是在内核中完成的，因为内核不知道库这种概念。在内核看来，所有程序都是可执行的代码段，有的代码段是可以映射和重定位的。

执行外部库搜索和加载的程序称为加载器，ELF 格式的是 ld-linux.so，a.out 格式的是 ld.so。而由于加载器可能有多种实现，也可能有多个版本，所以每个二进制文件中都需要指明使用哪个加载器，指明使用哪个加载器的功能就是用 segment 实现的，这种 segment 就是 PT\_INTERP 类型。由于 Golang 一般使用静态链接，所以你会发现几乎只有 Golang 的 ELF 格式中没有 PT\_INTERP 类型的 segment。这是其中一个 segment。事实上，所有让内核加载 ELF 文件时所提供给内核的信息都是以 segment 的形式存在的，内核只需要使用 segment 完成从磁盘到内存的映射加载工作即可。如图 10-5 和图 10-6 所示。Golang 的测试代码如下。

```
test.go
package main
func main() {
}
```

C 语言测试代码如下。

```
test1.c
int main(){
}
```



```

archerbroler@ubuntu:~$ readelf -l test
Elf file type is EXEC (Executable file)
Entry point 0x44e020
There are 7 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
-----
FileSiz        MemSiz             Flags             Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
R             1000
NOTE           0x0000000000000fc8 0x0000000000400fc8 0x0000000000400fc8
R             4
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
R E           1000
LOAD           0x0000000000004f00 0x000000000044f000 0x000000000044f000
R             1000
LOAD           0x0000000000006410a 0x000000000006410a 0x000000000006410a
R             1000
LOAD           0x000000000000b4000 0x000000000004b4000 0x000000000004b4000
RW           1000
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
RW           8
LOOS+5041580   0x0000000000000000 0x0000000000000000 0x0000000000000000
8

```

图 10-5

```

archerbroler@ubuntu:~$ readelf -l test1
Elf file type is EXEC (Executable file)
Entry point 0x4003e0
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
-----
FileSiz        MemSiz             Flags             Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
R             8
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
R             1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
R E           200000
LOAD           0x000000000000069c 0x000000000000069c 0x000000000000069c
R E           200000
LOAD           0x0000000000000e10 0x00000000000600e10 0x00000000000600e10
RW           200000
DYNAMIC        0x0000000000000e28 0x00000000000600e28 0x00000000000600e28
RW           8
NOTE           0x00000000000001d0 0x00000000000001d0 0x00000000000001d0
RW           8
NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
R             4
GNU_EH_FRAME   0x0000000000000574 0x0000000000400574 0x0000000000400574
R             4

```

图 10-6

一个程序一般会有 .dynamic section，而这个段就是放在类型为 PT\_DYNAMIC 的 segment 中的。这个 section 包括 segment 也都是用来服务于动态加载的。我们使用 ldd 命令可以读取到一个二进制依赖的库，此依赖关系就是写在 .dynamic section 中的。然而使用 ldd 命令显示的结果还包含了间接依赖的库。如图 10-7 所示。

```

archerbroler@ubuntu:~$ ldd /bin/ls
linux-vdso.so.1 => (0x00007ffff6c7cc000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f232a857000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f232a48e000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f232a21d000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f232a019000)
/lib64/ld-linux-x86-64.so.2 (0x000055a6e7022000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f2329dfc000)

```

图 10-7

也就是说, `.dynamic section` 记录了当前 ELF 执行需要的库的名字。至于到哪里去找这些库, 就是 `ld-linux.so` 的事情了。我们可以看到通过 `segment` 指定的这个闭环: `PT_INTERP` 指定了加载器, `PT_DYNAMIC` 指定了需要的库, 而这些需要的库又是通过加载器去实际加载的, 而加载的路径是操作系统提供的, 并不是内核提供的。一般是通过 `ldconfig` 程序生成的 `cache` 文件来缓存的, 并且操作系统在 `shell` 的层面提供了 `LD_LIBRARY_PATH` 这种可以方便指定库目录的机制。除此之外, 在 ELF 文件层次还提供了 `LD_PRELOAD` `segment` 来指定预加载的库。

```
RTLDLIST="/lib/ld-linux.so.2 /lib64/ld-linux-x86-64.so.2
/libx32/ld-linux-x32.so.2"

...
elif test -r "$file"; then
    RTLD=
    ret=1
    for rtld in ${RTLDLIST}; do
        if test -x $rtld; then
            dummy=`$rtld 2>&1`
            if test $? = 127; then
                verify_out=`$rtld --verify "$file"`
                ret=$?
                case $ret in
                    [02]) RTLD=$rtld; break;;
                esac
            fi
        fi
    fi
done
```

上述代码是从 `ldd` 中截取的, `ldd` 是一个脚本文件 (使用 `export LD_DEBUG=libs` 可以获得更详细的列表)。可以看到 `ldd` 实际调用了加载器完成的库依赖搜索。而我们也可以自己使用加载器执行, 如图 10-8 所示。

```
archer@ubuntu:~$ /lib64/ld-linux-x86-64.so.2 --list /bin/ls
linux_vdso.so.1 => (0x00007fffa91c0000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f201ccaf000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f201c8e6000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f201c675000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f201c471000)
/lib64/ld-linux-x86-64.so.2 (0x000055f8f1153000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f201c254000)
```

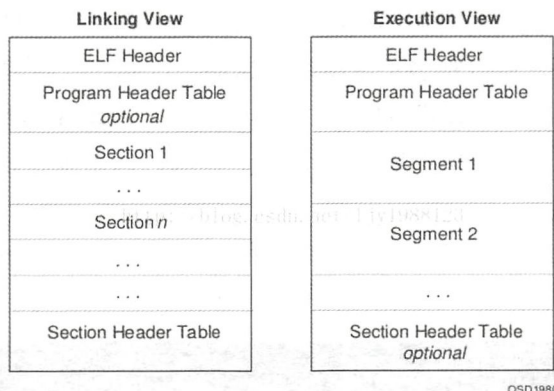
图 10-8



这样也能够得出库依赖。PT\_NOTE 则是记录程序的一些辅助信息。比如程序的类型、程序的所有者、程序的描述。这些信息不参与程序的执行，只有描述作用。但是这个段却经常被黑客看上。PT\_LOAD 就是真正的程序存储的地方。这个 segment 构成了程序的主体。

## 2. section 和 header

而 section 就是 segment 里面具体组织数据的格式了。每个 section 都有名字，这个名字是编译器给起的，你也可以自定义名字。链接器和加载器共同识别一些 section，所以可以进行约定好的操作。例如加载器看到.text 段就知道是代码段，而这个.text 段的创作者则是链接器。如图 10-9 所示，同一个 ELF 文件，连接器关心的内容和加载器关心的内容是不一样的。



OSD1980

图 10-9

我们使用 `readelf -h /bin/ls` 命令可以查看到一个典型的头部，如图 10-10 所示。这个头部里的 program headers 就是 segment 列表。可以看到 ELF 的头部的尺寸是 64 字节，所以 program headers 的起始地址从 64 字节的文件偏移开始，也就是紧挨着 ELF 的头部，执行的时候只关心 program headers。头部指明有 9 个 program header，每个 program header 的尺寸是 56 字节。有 29 个 section，每个 section 的尺寸是 64 字节。但是我们可以发现 program headers 和 section headers 中间会有不小的缝隙，这里的缝隙就是每一个 section 表的具体数据了，同时也是每一个 segment 的具体数据。因为 segment 和 section 是映射关系，它们共享这一大块数据，但是它们对于这个数据的认知角度不同。而 segment table 位于这段数据的前面，section table 位于这段数据的后面。



```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
  Class:                           ELF64
  Data:                               2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                               EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:                http://blog.csdn.0x4049a03
  Start of program headers:           64 (bytes into file)
  Start of section headers:          124728 (bytes into file)
  Flags:                               0x0
  Size of this header:                64 (bytes)
  Size of program headers:            56 (bytes)
  Number of program headers:           9
  Size of section headers:            64 (bytes)
  Number of section headers:          29
  Section header string table index: 28

```

图 10-10

ELF 文件规定在执行的时候不需要 section table，而大部分的二进制工具却都是依赖 section table 的，所以很多不希望被调试的二进制在发布的时候都会将 section table 去掉，这是俗称的“strip”的其中一步。去掉了 section table 后，section 依然存在，也有工具能从 section 中恢复出 section table，但是难度比较高。这就是攻与防的较量了。

通过 file 命令就能够发现系统上自带的 ls 命令是已经经过 strip 的了，如图 10-11 所示。

```

archerbroler@ubuntu:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=eca98eeadafdff44caf37ae3d4b227132861218, stripped

```

图 10-11

我们继续通过 readelf -l /bin/ls 命令观察二进制的 segment 的细节。如图 10-12 所示都是不带 PT 前缀的，第一个 segment 永远是 PHDR，因为这个 segment 是用来说明 program headers 位置的，虽然在头部有指定在文件中的偏移，但是并没有指定这个头部放在内存的哪个地方。所有在 segment 头部的条目都既有文件地址又有内存地址。值得注意的是，它还有物理地址，这个物理地址只在某些机器上有效，大部分的机器都是直接用了 virtaddr，并且 systemv 格式的 ABI 根本不识别物理地址。GNU\_STACK 表示的是我们的栈，重要的是它具有 RW 权限，所以我们知道这个程序的栈没有可执行权限。

如果你用 exestack -s /bin/ls 命令就会发现这个 segment 有执行权限，变成 RWE 了。现代的编译器默认都不会给栈执行权限，如果发现了二进制的栈有了执行权限，

那可能就有安全问题了。如图 10-13 和图 10-14 所示。

```
Program Headers:
```

Type	Offset	FileSiz	VirtAddr	PhysAddr
			MemSiz	Flags Align
PHDR	0x0000000000000040	0x0000000000000040	0x0000000000000040	0x0000000000000040
	0x00000000000001f8	0x00000000000001f8		R E 8
INTERP	0x0000000000000238	0x0000000000000238	0x0000000000000238	0x0000000000000238
	0x000000000000001c	0x000000000000001c		R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]				
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000001dae4	0x0000000000001dae4		R E 200000
LOAD	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00
	0x0000000000000800	0x00000000000001568		RW 200000
DYNAMIC	0x0000000000001de18	0x0000000000001de18	0x0000000000001de18	0x0000000000001de18
	0x00000000000001e0	0x00000000000001e0		RW 8
NOTE	0x0000000000000254	0x0000000000000254	0x0000000000000254	0x0000000000000254
	0x0000000000000044	0x0000000000000044		R 4
GNU_EH_FRAME	0x0000000000001a654	0x0000000000001a654	0x0000000000001a654	0x0000000000001a654
	0x000000000000080c	0x000000000000080c		R 4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000		RW 10
GNU_RELRO	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00
	0x0000000000000200	0x0000000000000200		R 1

图 10-12

```
archerbroiler@ubuntu:~$ execstack -s ./ls
```

图 10-13

```
archerbroiler@ubuntu:~$ readelf -l ./ls
```

```
Elf file type is EXEC (Executable file)
Entry point 0x4049a0
There are 9 program headers, starting at offset 64
```

```
Program Headers:
```

Type	Offset	FileSiz	VirtAddr	PhysAddr
			MemSiz	Flags Align
PHDR	0x0000000000000040	0x0000000000000040	0x0000000000000040	0x0000000000000040
	0x00000000000001f8	0x00000000000001f8		R E 8
INTERP	0x0000000000000238	0x0000000000000238	0x0000000000000238	0x0000000000000238
	0x000000000000001c	0x000000000000001c		R 1
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]				
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000001dae4	0x0000000000001dae4		R E 200000
LOAD	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00
	0x0000000000000800	0x00000000000001568		RW 200000
DYNAMIC	0x0000000000001de18	0x0000000000001de18	0x0000000000001de18	0x0000000000001de18
	0x00000000000001e0	0x00000000000001e0		RW 8
NOTE	0x0000000000000254	0x0000000000000254	0x0000000000000254	0x0000000000000254
	0x0000000000000044	0x0000000000000044		R 4
GNU_EH_FRAME	0x0000000000001a654	0x0000000000001a654	0x0000000000001a654	0x0000000000001a654
	0x000000000000080c	0x000000000000080c		R 4
GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	0x0000000000000000
	0x0000000000000000	0x0000000000000000		RWE 10
GNU_RELRO	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00	0x0000000000001de00
	0x0000000000000200	0x0000000000000200		R 1

图 10-14

我们也能在这个命令的下方发现 section 到 segment 的映射表。仔细观察 segment 表会发现有两个连续的 LOAD segment，编号是 02 和 03，在 section 的映射表里观察 02、03 编码，可以发现两者存储的 section 并不相同。典型的存储数据 .data、.bss 等在 03，而存储代码的 .text 在 02。程序在启动时，内核首先加载 LOAD segment 的



内容到内存中,然后用 PT\_INTERP 指定的加载器加载 LD\_PRELOAD 和 DYNAMIC segment 中指定的库到内存中,并且对这些库进行初始化,就是调用库的 INIT segment (.init section) 中的逻辑。segment 和 section 的对应关系,如图 10-15 所示。

```

Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .re
la.plt .init .plt .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
03 .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04 .dynamic
05 .note.ABI-tag .note.gnu.build-id
06 .eh_frame_hdr
07
08 .init_array .fini_array .jcr .dynamic .got

```

图 10-15

这些 section 具体的用途如果自己动手写一下链接脚本就比较容易理解了。如图 10-16、图 10-17 和图 10-18 所示是 readelf -S /bin/ls 的部分结果,首先我们看到了一系列的 section,先不去关心每个 section 的意义。我们需要知道现在观察的是一个可执行文件,但不只是可执行文件才具有 ELF 格式,静态库、动态库,甚至编译中间的.o 文件也都是 ELF 格式的。但是例如.o 格式的中间编译文件是没有经过链接步骤的,所以它的很多 section 的 address 会是 0,经过链接之后才会有真实的赋值,并且所拥有的 section 的种类一般也是有区别的。每个 section 的 offset 就表明了它们具体的 section 数据在文件中的偏移,都是位于 segment table 和 section table 之间的。

```

There are 29 section headers, starting at offset 0x1e738:

Section Headers:
[Nr] Name              Type              Address            Offset
Size              EntSize          Flags Link Info Align
[ 0] 0000000000000000 NULL              0000000000000000 00000000
[ 1] .interp              PROGBITS          0000000000400238 00000238
000000000000001c 0000000000000000 A 0 0 1
[ 2] .note.ABI-tag        NOTE              0000000000400254 00000254
0000000000000020 0000000000000000 A 0 0 4
[ 3] .note.gnu.build-id   NOTE              0000000000400274 00000274
0000000000000024 0000000000000000 A 0 0 4
[ 4] .gnu.hash            GNU_HASH          0000000000400298 00000298
00000000000000c0 0000000000000000 A 5 0 8
[ 5] .dynsym              DYNSYM            0000000000400358 00000358
00000000000000d8 0000000000000018 A 6 1 8
[ 6] .dynstr              STRTAB            0000000000401030 00001030
000000000000005dc 0000000000000000 A 0 0 1
[ 7] .gnu.version          VERSYM            000000000040160c 0000160c
0000000000000112 0000000000000002 A 5 0 2
[ 8] .gnu.version_r        VERNEED           0000000000401720 00001720
0000000000000070 0000000000000000 A 6 1 8
[ 9] .rela.dyn             RELA              0000000000401790 00001790
00000000000000a8 0000000000000018 A 5 0 8
[10] .rela.plt             RELA              0000000000401838 00001838
00000000000000a0 0000000000000018 A 5 24 8
[11] .init                PROGBITS          00000000004022b8 000022b8
000000000000001a 0000000000000000 AX 0 0 4
[12] .plt                  PROGBITS          00000000004022e0 000022e0
00000000000000710 0000000000000010 AX 0 0 16

```

图 10-16



[13]	.plt.got	PROGBITS	00000000004029f0	000029f0
	0000000000000008	0000000000000000	AX 0 0	8
[14]	.text	PROGBITS	0000000000402a00	00002a00
	0000000000011289	0000000000000000	AX 0 0	16
[15]	.fini	PROGBITS	0000000000413c8c	00013c8c
	0000000000000009	0000000000000000	AX 0 0	4
[16]	.rodata	PROGBITS	0000000000413ca0	00013ca0
	00000000000069b4	0000000000000000	A 0 0	32
[17]	.eh_frame_hdr	PROGBITS	000000000041a654	0001a654
	000000000000080c	0000000000000000	A 0 0	4
[18]	.eh_frame	PROGBITS	000000000041ae60	0001ae60
	00000000000002c84	0000000000000000	A 0 0	8
[19]	.init_array	INIT_ARRAY	000000000061de00	0001de00
	0000000000000008	0000000000000000	WA 0 0	8
[20]	.fini_array	FINI_ARRAY	000000000061de08	0001de08
	0000000000000008	0000000000000000	WA 0 0	8
[21]	.jcr	PROGBITS	000000000061de10	0001de10
	0000000000000008	0000000000000000	WA 0 0	8
[22]	.dynamic	DYNAMIC	000000000061de18	0001de18
	000000000000001e0	0000000000000010	WA 6 0	8
[23]	.got	PROGBITS	000000000061dff8	0001dff8
	0000000000000008	0000000000000008	WA 0 0	8
[24]	.got.plt	PROGBITS	000000000061e000	0001e000
	0000000000000398	0000000000000008	WA 0 0	8
[25]	.data	PROGBITS	000000000061e3a0	0001e3a0
	0000000000000260	0000000000000000	WA 0 0	32
[26]	.bss	NOBITS	000000000061e600	0001e600
	00000000000000d68	0000000000000000	WA 0 0	32
[27]	.gnu_debuglink	PROGBITS	0000000000000000	0001e600
	00000000000000034	0000000000000000	0 0 0	1
[28]	.shstrtab	STRTAB	0000000000000000	0001e634
	0000000000000102	0000000000000000	0 0 0	1

图 10-17

```

key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

```

图 10-18

读取每一个 section 的内容时，readelf 一般会提供常用的选项，例如 readelf -r /bin/ls 或者 readelf -d /bin/ls 等都可以读取到具体的 section 的内容。这个 section table 在执行的时候是不会被加载到内存中的，因为加载器和内核都是识别 segment table 的。

ELF 文件中有两种类型的 section，即可分配的和不可分配的，可分配的 section 就是指在运行时会被加载到内存，不可分配的 section 就是给调试器用的，在执行的时候没用。Strip 程序可以把这些在执行期无用的东西删除，使得文件更小。有两种符号表，即 .symtab 和 .dynsym，.dynsym 是 .symtab 的子集，.dynsym 是程序运行期需要的，而 .symtab 是在程序调试时需要的。strip 可以把 .symtab 去掉。

很多 section 并不是存储的字符串，而是存储的字符串索引。而字符串是存储在单独的地方的。比如在 ELF 的文件头部就有 e\_shstrndx 域，其用于表示 section table 的各个条目所对应的字符串的实际存储地址，而在实际的 section table 里却全部是数字地址和数字索引，通过这个索引就可以在 e\_shstrndx 所对应的 section 字符串表中找到对应的字符串了。

### 3. 符号和字符串

符号和字符串是不一样的。符号表示的是程序里的函数名或者变量名，而字符串不但包括符号，还包含代码里出现的字符串常量。

我们会发现有两个表示符号的 section，一个是 `.dynsym`；另外一个 `.symtab`。`.symtab` 中包含了 `.dynsym`，但是 `.dynsym` 仍有必要存在的原因是，这部分的符号是运行时所需要的，而 `.symtab` 中的其他符号运行时不需要。所以很多二进制发布的时候会去掉 `.symtab` section，使得调试变得困难或者减小二进制大小。符号表可能给大家的印象就是放了很多字符串，符号表条目实际上有很多种，字符串只是其中的一种，并且即使是字符串，也不是直接放在符号表里的，而是符号表里存放了字符串表的索引，真正的字符串放在字符串表里。这也并不是说 `.dynsym` 会永远存在，在编译时指定 `-static` 和 `-nostdlib` 使用的时候，由于没有外部的访问也就不需要解析外部的符号，所以 `.symtab` 也会不存在。

对应的字符表也有两个 section：一个是 `.strtab`；另一个是 `.dynstr`。而字符串表又不同于存放符号的字符串，代码里出现的字符串才是主要的构成。

### 4. 强符号和弱符号

`readelf --dyn-syms` 命令可以看到动态符号表，这个符号表里面包含了所有来自 `rela.dyn` 和 `rela.plt` 中所需要的符号。如果你使用 `readelf -S /bin/ls` 命令查看到 `.dyn.sym` 的 section 编号是 4，然后使用 `readelf -p 4` 命令查看具体内容，你将得到乱码。大部分 section 的存储都不是直接的存储字符串，而是有内定的二进制结构体的直接输出。所以 `readelf` 实现了一些常用的选项，用于解析这些二进制的 section。典型的观察符号的选项是 `-s`，可以列出所有的符号，也就是 `.symtab` section 之中的符号（这里面包含了 `.dynsym` 的所有条目）。当然也可以只查看 `.dynsym` 中的符号，就是使用 `--dyn-syms` 选项。典型的区别是你会在 `.symtab` 中发现大量的 LOCAL 范围的符号，而在 `.dynsym` 中则发现不到，如图 10-19 所示。

```

1683: 0000000000064088 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1684: 0000000000064090 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1685: 0000000000064098 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1686: 00000000000640a0 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1687: 00000000000640a8 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1688: 00000000000640b0 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1689: 00000000000640b8 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1690: 00000000000640c0 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1691: 00000000000640c8 8 OBJECT LOCAL DEFAULT 15 _ZN5boost3log12v2s_mt_pos
1692: 00000000000640d0 4 OBJECT LOCAL DEFAULT 15 _ZN6google8protobuf9kint
1693: 00000000000640d4 4 OBJECT LOCAL DEFAULT 15 _ZN6google8protobuf9kint
1694: 00000000000640d8 8 OBJECT LOCAL DEFAULT 15 _ZN6google8protobuf9kint

```

图 10-19

其他的 3 种符号类型是 UNIQUE、GLOBAL、WEAK。其中 UNIQUE 是 GCC 定义的 ELF 格式的扩展。最重要的是 GLOBAL 和 WEAK，就是我们常说的强符号和弱符号。

我们经常在编程中碰到符号重复定义的错误，这是因为在多个目标文件中含有相同名字全局符号的定义。比如我们在目标文件 A 和目标文件 B 中都定义了一个全局整形变量 foo，并将它们都初始化，那么链接器将 A 和 B 进行链接时会报错，如下所示。

```
b.o:(.data+0x0): multiple definition of `foo'
a.o:(.data+0x0): first defined here
```

这种符号的定义可以被称为强符号，上面举的是变量的例子，符号也一样，而有些符号的定义可以被称为弱符号（Weak Symbol）。

对于 C/C++ 语言来说，编译器默认函数和初始化了的全局变量为强符号，未初始化的全局变量为弱符号。也可以通过 GCC 的“\_\_attribute\_\_((weak))”修饰语法来定义任何一个符号为弱符号。

针对强弱符号的概念，链接器会按如下规则选择与处理被多次定义的全局符号。

- 规则 1：不允许强符号被多次定义。
- 规则 2：如果一个符号在某个目标文件中是强符号，在其他文件中都是弱符号，那么选择强符号。
- 规则 3：如果一个符号在所有目标文件中都是弱符号，那么选择其中占用空间最大的一个符号定义。

强弱符号在我们正向编程的时候尽量不要涉及，因为很容易导致出现管理上的问题。而由于多个强符号的出现是不被允许的，所以如果在程序中定义了与库中的强符号冲突的符号，链接时就会报错。但是很多库在设计的时候有允许让用户覆盖自己的定义的需求，例如覆盖定义 dl\_discover\_osversion 函数就可以欺骗编译器，伪造操作系统的版本。在逆向的破解或者研究的时候，也可以方便地调整符号的强弱来在二进制层面打开或者关闭某个特定的符号定义。这些功能是通过强引用与弱引用实现的。

当一个库的一个函数被定义为弱引用（“\_\_attribute\_\_((weakref))”）时，它就可以被强引用覆盖，而我们平时所定义的符号都是默认为强引用的。对于指定的弱引用符号，链接器即使没有找到也不会报错，而是默认其为 0，但是如果你是在你的



程序中使用了没有定义的弱引用，则一定无法运行，因为运行时找不到这个符号的具体定义。所以弱引用几乎是库的专利，程序中可以通过覆盖定义这个弱引用来取代库中的定义。而还有一种用法就是在库中某个符号虽然定义为强引用，但是可以在程序中定义相同的符号为弱引用，在链接的时候有不同的版本。

例如，如果一个程序被设计成可以支持单线程或多线程的模式，就可以通过弱引用的方法来判断当前的程序是链接到了单线程的库还是多线程的库。如果在编译时有“-lpthread”选项，就能够决议到符号，从而执行 pthread 库中的多线程版本的程序，反之就链接不到外部符号，就会使用内部的单线程版本的程序。例如，我们可以在程序中定义一个 pthread\_create 函数的弱引用，然后程序在运行时动态判断是否链接到 pthread 库，从而决定是执行库中的版本，还是执行程序中定义的版本。

这几种 section 之间的关系和它们分别的作用是 ELF 文件中相对最难理解的知识点，也是动态符号解析的关键所在，同时还是很多安全问题的高发地。大部分的技术都是被需求驱动的。所以理解一个技术，首先要理解这个技术的需求。将 ELF 标准的表设计复杂的最重要的原因是服务于 PIC，就是与位置无关的编译，然后是延迟绑定技术，延迟绑定技术可以让符号在用到的时候才有必要解析。

其实 dynamic 相当于运行时的加载器的数据库，加载器从这个 section 获得所需的外部库，该 ELF 文件符号表 (.dynsym) 和字符串表 (.dynstr) 的地址；额外的动态库搜索路径；初始化和结束代码 (.init 等) 的地址；动态链接的重定向 section (.rela.dyn 等，rela.dyn 和 rel.dyn 是指同一个东西) 地址；.got.plt 等我们完全可以从 section header 中获得信息。ELF 标准之所以要在这里再提供一遍这些定义是因为不是每个 ELF 文件都有 section table，而这些 section 又是加载器所必需的，所以也并不是说 section 在运行的时候就是无用的。

我们使用 ldd /bin/ls 命令，输出结果如图 10-20 所示。

```
root@ubuntu:~/sdb1/yunweishi/build/pragram/elfscan# ldd /bin/ls
linux-vdso.so.1 => (0x00007ffc5f7000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f7d7f324000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f7d7ef5b000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f7d7ecea000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f7d7eae6000)
/lib64/ld-linux-x86-64.so.2 (0x000055be70b79000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f7d7e8c9000)
```

图 10-20

这些是 ldd 程序通过解析符号表和实际运行之后获得的其所需要的外部库的列表。我们使用 readelf --dyn-syms /bin/ls 命令就能看到哪些符号需要被从外部解析，

而 `ldd` 的运行就相当于实际试图去解析执行二进制，并且还处理了解析的符号需要的其他外部库符号的情况。

链接的时候即使不使用 PIC，在技术上也是可行的，大不了所有的库都固定地址。然而 x86 的这种模式很快就会用光地址空间，并且这种模式就需要存在一个统一的地址分配机构，这显然不是分布式开发的 Linux 的风格。所以每一个库视情况映射到不同进程的不同地址几乎就是唯一的选择，虽然其仍然在物理内存中只保存一份，但由于库代码是只读的，并且 Linux 的映射支持一对多的映射，所以 Linux 就这么做了。

但是如果这么做就需要面临一个问题，就是一些外部库的符号的地址一定是要在运行时填充的，因为它们是不固定的。还有一个问题就是一定要用符号来作为程序和库之间的桥梁，因为在编译时只看到了用户使用了头文件中存在的符号，而不是一个数字的编码，如果使用数字编码，虽然能够加快查找速度，但是要求运行库和链接过程共享相同的数字编码。实际上这也是可以实现的，`.gnu.hash` section 就是用于将字符串哈希成数字。而之所以一定要保留动态链接的符号，有一个重要的原因是我们观察每个符号实际在二进制中的存储方式，例如 `sigprocmask@GLIBC_2.2.5` (3)，都要指定库的名字和版本，而这些指定只能使用字符串，否则又该需要一个集中式的控制中心了，所以在符号查找时使用字符串不可避免，但是一定有更高效率的解决方案。如果在查找过程中仍然使用字符串，因此而牺牲的性能就是易用性与高效性。

对于非 PIC 程序来说，当我们使用 `readelf -l /bin/ls` 命令时，会发现它的 `VirtAddr` 和 `PhysAddr` 两个域都是有具体值的。但是如果查看一个库，例如 `readelf -l /usr/lib/libcrypto.so.1.1`，它的 `VirtAddr` 和 `PhysAddr` 都会是 0（以第一个为准，`.text` 代码段位于第一个 LOAD），这些库代码内部就是使用偏移而不是绝对地址。

要确定这些符号的具体地址，有两种类型，一个是数据；另一个是函数。所以你会发现 `.rela.dyn` 和 `.rela.plt`，`.rela.plt` section 是用于函数重定位的；`.rela.dyn` section 用于变量重定位。还有 `.got` 和 `.got.plt`，同样也是前者用于数据重定向，后者用于函数重定向。这两者的命名规则不一样，很容易导致使用上的迷惑。而我们也很容易产生疑问，为什么会有两组用于解析外部符号的段？是为了满足另外一个需求——延迟绑定。如图 10-21 和图 10-22 所示。

Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
	FileSiz	MemSiz			
PHDR	0x0000000000000040	0x0000000000400040	0x0000000000400040		
	0x00000000000001f8	0x00000000000001f8	R E	8	
INTERP	0x0000000000000238	0x0000000000400238	0x0000000000400238		
	0x000000000000001c	0x000000000000001c	R	1	
[Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]					
LOAD	0x0000000000000000	0x0000000000400000	0x0000000000400000		
	0x0000000000001dae4	0x000000000001dae4	R E	200000	
LOAD	0x0000000000001de00	0x000000000001de00	0x000000000001de00		
	0x0000000000000800	0x0000000000001568	RW	200000	

图 10-21

Program Headers:					
Type	Offset	VirtAddr	PhysAddr	Flags	Align
	FileSiz	MemSiz			
LOAD	0x0000000000000000	0x0000000000000000	0x0000000000000000		
	0x000000000000262c8c	0x000000000000262c8c	R E	200000	
LOAD	0x000000000000262da8	0x000000000000462da8	0x000000000000462da8		
	0x000000000000265c8	0x00000000000029db8	RW	200000	

图 10-22

## 5. 动态库

动态库的核心包含了两个层次的代码共享：编译成的二进制可以不用每个二进制文件都包含一份动态库的拷贝；在执行的时候，动态库的代码段只需要加载一次，后面再有人用到同一个动态库，内核就可以把动态库代码段加载到的页面直接映射到其他需要的进程中，这样代码段也不用加载多次（代码段是只读的）。

由于 CPU 执行的时候必须要使用相对或者绝对地址，虽然代码段的每个函数物理地址是一样的，但是它们映射到每个进程内存空间的地址都是不一样的。所以动态库需要有一份符号表，记录其在代码段的偏移，还需要一个全局偏移，意味着其整个符号表在进程地址空间中的偏移。

一个使用了动态库的可执行文件，其内部有调用这些符号的代码，这种代码无法在链接期解析出具体的地址偏移（因为链接器根本没有实际链接它们），所以它们在二进制文件中只是一个占位符（rela.dyn、rela.plt），其地址需要在动态库加载了之后再填充。而这种调用是分散在整个程序中的，所以在加载后就得去搜索并找到所有的未解析符号去解析它们，这样肯定是不合适的，所以需要有一个表，记录所有这些没有解析的符号（.dynsym）。

动态库在内存中只要执行到任何一行动态库的代码，动态库就可以通过偏移找到本库内的其他符号，因为同一个库的符号偏移，本库内的函数都是知道的。但可惜的是 i386 不支持通过当前执行指令（PC）偏移的寻址方式（如果支持就简单了，根本不需要重分配，只需要在执行到的代码中使用偏移就可以了）。但是 x64 是支持



PC 的，所以 ELF 在 x64 时代有点变化。

ELF 文件链接完成，将调用动态库的符号放到这些表里。当动态库加载的时候，加载器要负责查找这个表，将加载的动态库的对应的符号所在内存的地址填充到可执行文件的这些表（.got）中，如此完成加载时候的符号绑定。同时解决了动态库位置不固定的问题。

## 6. 动态库装载

程序在执行的过程中，可能有些引入的 C 库函数到结束时都不会被执行。所以 ELF 采用延迟绑定技术，在第一次调用 C 库函数时才会去寻找真正的位置进行绑定。但是也有设置为加载的时候就全部完成绑定（RELRO 攻击对抗技术就是这样）。延迟绑定技术，如图 10-23 所示。

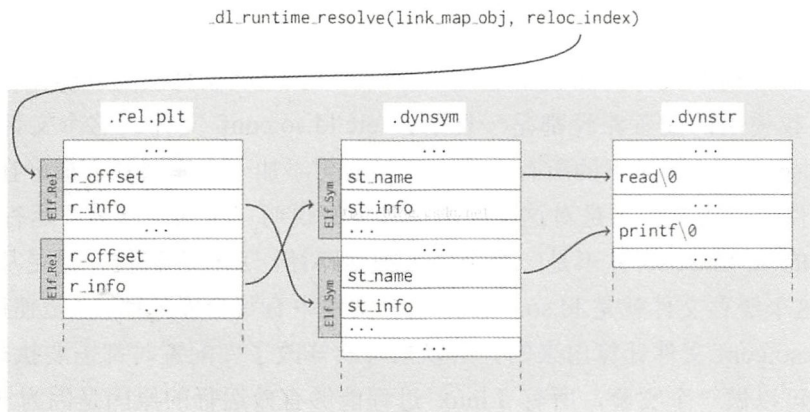


图 10-23

一个应用由一个主要 ELF 二进制文件（可执行文件）和数个动态库构成，它们都是 ELF 格式的。每个 ELF 对象由多个 segments 组成，每个 segment 则含有一个或多个 sections。这些段看起来很多，但是大多数都非常简单。每一个段基本只存储一种类型的数据，例如 `.dynstr` 里面就只有字符串。

例如 `.rel.plt` 中有待解析外部符号的桩函数。每个 ELF 要访问外部符号的时候，首先进入到 `.rel.plt` 中对应的桩函数，这个桩函数会进入对应的 `.got.plt` 中的条目来加载对应的外部符号，并且把符号地址存放在 `.got.plt` 中，这样以后再访问这个符号的时候，`.rel.plt` 中的桩函数就可以直接从 `.got.plt` 中拿。这就是惰性加载的原理。

而每个 .rel.plt 和 .rel.dyn 中的条目都指向一个 .dynsym 条目, 每个 .dynsym 条目都指向一个 .dynstr 条目。 .dynstr 里面只有字符串, 而 .dynsym 中存储的数据有这个符号的虚地址 (函数在没有被执行的时候, 虚地址自然为 0) 和符号的类型, 以及绑定类型, 如图 10-24 所示。

```
Symbol table '.dynsym' contains 5 entries:
Num:      Value              Size Type      Bind   Vis      Ndx Name
0: 0000000000000000      0 NOTYPE    LOCAL DEFAULT UND
1: 0000000000000000      0 FUNC     GLOBAL DEFAULT UND puts@GLIBC_2.2.5 (2)
2: 0000000000000000      0 FUNC     GLOBAL DEFAULT UND __libc_start_main@GLIBC_2.2.5 (2)
3: 0000000000000000      0 NOTYPE    WEAK   DEFAULT UND __gmon_start__
4: 0000000000000000      0 FUNC     GLOBAL DEFAULT UND perron@GLIBC_2.2.5 (2)
```

图 10-24

### 10.3.3 进程加载器

前面说过 ELF 文件的加载器是 ld-linux.so, 而 a.out 文件的加载器是 ld.so。但是这两个加载使用的配置路径都是一样的: /etc/ld.so.conf 文件。这个文件里一般是 include ld.so.conf.d 目录下的所有文件, 所以要想添加一个库路径, 最好在目录下建立一个文件。因为文件名是对这个库用途的良好说明。添加完就需要运行 ldconfig, 因为实际的 ld-linux.so 并不是一个个地去搜索路径, 那样会极慢, 而是从缓存中直接查询。这个缓存文件就是 ld.so.cache, 这个文件中有每个库的路径, 是使用 ldconfig 程序和 ld.so.conf 文件计算出来的。所以当每次修改了库配置时都需要执行此命令。

你也可以做一个实验, 所有 Linux 进程能够有效运行的原因是因为 ld-linux.so 位于同样的目录/lib/目录下。如果这个文件被移动或者重命名, 几乎所有的程序都不能被执行 (用 Golang 编译的, 不使用 ld-linux.so 加载的程序仍可以执行)。此时如果你想恢复执行, 就得将 ld-linux.so 继续拷贝到/lib/目录下, 然而你会发现 mv 命令也无法执行了。但是 builtin 的 cd 之类的命令却可以执行。恢复的办法是显式地使用 ./ld-linux.so mv a b, 当然还要加上必要的参数。这里只是要论证一点: 如果要执行所有 GCC 编译的进程, 其本质上是加载器程序先执行, 然后由加载器加载实际的二进制进程进行执行。就好像 Python 程序无法直接执行, 但是经过 shell 的设置后就可以自动找到 Python 程序来执行了。

另外, 同一个库可能有多个版本, 但这是不冲突的, 只要你将路径都加入即可。

### 10.3.4 链接与执行

ELF 文件由 ld 程序连接，由 ld.so 程序执行，分为静态和动态两个过程。同样地，在二进制文件中 segment 是动态的文件布局，由 ld.so 使用。而 section 是静态的文件布局，由 ld 使用。

一般编译代码和执行代码的是不同的人，不同的机器。所以静态编译就需要一种语言告诉动态编译怎么执行，这个语言就是 ELF 文件格式规范。

ELF 文件里面有 segment 和 section，segment 中包含 section。我们知道在程序执行的时候，所有的应用程序都是首先通过加载器 ld.so 加载到内存然后执行的（内部集成了加载器的除外），而所有的 ELF 文件也都是经过链接的过程而形成的。Segment 提供信息给 ld.so 这个加载器，告诉它如何加载，而 section 提供 ld 程序，告诉它如何链接。

其实 section 本质上就是起到记录的作用，ELF 没有它也完全可以设计成正常工作模式，因为 segment 已经提供了工作所需要的信息。但是 section 是在 ld 链接时的工作过程，记录了将不同的内容放到不同的文件位置的分布情况，section 表就是这个分布的一个总体描述。section 的最大意义在于让 ELF 有语义的意义。如果没有 section 只有 segment，ELF 就只是个可以执行的文件，别人没办法分析它的组成和它的二进制格式。由于缺少了二进制层次上的语义，也正因为决定要在二进制文件中保留 section 了，那么有的 segment 也就可以用 section 来组织。毕竟可执行文件的物理组装是通过 section 完成的，而 segment 也要告诉 ld.so 怎么使用这个物理文件，所以两者发生交互就会方便很多。而现在的很多 section 在运行期间也是有用的，例如 .text, .got 可以用来找到动态库的符号。

也正是由于一些 section 是可有可无的，所以在 Linux 系统下的 strip 程序专门用来将一些没用的段去除以减小文件的大小。strip 命令从文件中有选择地除去行号信息、重定位信息、调试段、typchk 段、注释段、文件头以及所有或部分符号表。一旦用了该命令，则很难调试文件的符号。因此，通常应该只在已经调试和测试过的生成模块上使用 strip 命令。使用 strip 命令减少对象文件所需的存储量开销。strip 也可以手动指定你要删除的 section，使得大小优化更加激进。ld 命令的“-s”选项也有同样的功能。

一个重要的区别是 segment table 和 section table 之间的关系。确切地说，在当前的标准里运行期间不是不需要 section，而是不需要 section table。也就是说，如果没



有在 ld 阶段用到 section 和 section table, 链接的时候完全可以直接使用 segment 来组织数据, 但是可能就需要修改 ELF 标准了。考虑到是编译和运行是两个完全独立的过程, ELF 标准最后采用了两个维度组织 ELF 文件格式。

链接器常见的 c 级是 gnu linker, 默认是集成在 GCC 里面的, 但是谷歌觉得它太慢, 所以又开发了一个新的链接器——gold。Gnu linker 使用 bfd 库, 而 gold 只是部分支持 ELF 的全部特性, 去除了谷歌认为不需要的, 并且没有使用 bfd 库, 所以可以做到链接速度很快。gold 现在也被加入到 binutils 包里面了。

在 Linux 系统下链接程序以前只有 GCC 一整套工具, 后来越来越多的大厂进入, 例如谷歌开发了 gold 链接器 (已经被纳入 GCC 工具包, 使用 “-fuse-lld=gold” 编译选项启用), Intel 推出了 intel linker, clang 推出了 lld, 谷歌甚至直接去掉了加载器, 并且发明了 Golang。

但是无论各大厂商如何努力, 大部分公司由于人才知识的惯性, 大多数都还是在使用 ld。很多研究机构在使用 lld, 也有很多超级项目开始使用 gold, 后台开发也越来越多的使用 Golang。Ld 在逐渐丢失阵地。

GNU ld 在使用时确实不是非常友好, 使用它还需要掌握相当多的知识量。因为 ld 的链接脚本是为 COFF 设计的, 这个链接脚本的语法也是根据古老的 SVR3 演变的。GCC ld 完全围绕链接脚本设计, 所以几乎没办法改变。而 gold 在设计的时候就直接去掉了链接脚本, 采用链接器自动生成的方法, 类似 ld 的默认链接脚本。所以, 学习链接更重要的是学习链接的理论知识, 而并不是学习链接脚本工具本身。而链接脚本着实能够服务于高级用户, 这一点在加固病毒领域确实是必不可少的。

然而在实际的使用过程中, gold 在很多情况下也并不比 ld 快, 毕竟都只是工具, 能够最终生成想要的 ELF 文件即可。

Linux 系统下的 ld 命令在组织库的时候有一个常见的问题, 就是库出现的顺序。ld 命令从左向右解析, 左边的二进制需要的符号必须在右边有提供。所以大型项目组织库的顺序也是比较棘手的。一般用户自己实现的库放在前面, 系统库放在后面。有的库是交叉依赖的, 例如 liba 依赖 libb, libb 又反过来依赖 liba。这时就可以写两次 gcc main.c ./liba.a ./libb.a liba.a, 或者使用 ld 的 group 概念。

## 裸程序

一个程序从编译到执行, 除了约定的共同识别的二进制格式之外, 还需要有共同的接口。这个接口就是 \_start 函数。几乎所有使用 Linux 系统的机器都默认安装了 glibc 库, 而这个库实现了 C 语言标准的 main 函数, 而这个 main 函数却不是启动时

被最先调用的，链接器和加载器约定最先调用的函数是 `_start` 函数。无论是否使用 `glibc` 都是如此。我们平时编写程序直接写 `main` 函数，是因为 `glibc` 在自己的 `_start` 函数里调用了 `main` 函数。

你可以很容易测试出库里的调用情况，代码如下，编译如图 10-25 所示。

```
void main(){}
static inline void func2(){}

_start(){
    main();
    func2();
}

archerbroler@ubuntu:~$ gcc test1.c
test1.c:5:1: warning: return type defaults to 'int' [-Wimplicit-int]
_start(){
^
/tmp/cc12d4Qn.o: In function '_start':
test1.c:(.text+0xe): multiple definition of '_start'
/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
archerbroler@ubuntu:~$ gcc test1.c -nostdlib
test1.c:5:1: warning: return type defaults to 'int' [-Wimplicit-int]
_start(){
^
```

图 10-25

这种编译方法会使 GCC 自动链接它自己的库，由于在我们的程序中也定义了 `_start` 函数，所以就与 `glibc` 的定义冲突了。然而我们可以让 `gcc` 默认不去链接它自己的库。使用 “`-nostdlib`” 选项就可以做到，或者这里使用 “`-nostartfiles`” 选项就可以绕过 `glibc` 的 `_start` 函数。

然而如果你绕过 `glibc` 的 `_start` 函数，就得自己做 `glibc` 底层要做的事情了，大部分事情是与内存相关的，否则你的程序一运行就会 `coredump`。

我们也可以使用 `g++ -v test.cpp` 查看当前的详细 `ld` 参数，这样你就可以不使用 `g++` 的封装，而使用 `ld` 命令来直接链接程序了。用 `g++ -v test.cpp` 命令的输出结果，如图 10-26 所示。

```
/usr/lib/gcc/x86_64-linux-gnu/5/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/5/liblto_plugin.so -plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/5/lto-wrapper -plugin-opt=-fresolution=/tmp/ccQxwdrn.res -plugin-opt=-pass-through=lgcc_s -plugin-opt=-pass-through=lgcc -plugin-opt=-pass-through=lc -plugin-opt=-pass-through=lgcc_s -plugin-opt=-pass-through=lgcc --sysroot=/ --build-id --eh-frame-hdr -m elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -z relro /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crt1.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/5/crtbegin.o -L/usr/lib/gcc/x86_64-linux-gnu/5 -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../../lib -L/usr/lib/x86_64-linux-gnu -L/usr/lib/../../lib -L/usr/lib/gcc/x86_64-linux-gnu/5/../../../../tmp/ccwEWVIm.o -lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s -lgcc /usr/lib/gcc/x86_64-linux-gnu/5/crtend.o /usr/lib/gcc/x86_64-linux-gnu/5/../../../../x86_64-linux-gnu/crtn.o
```

图 10-26

### 10.3.5 ELF 文件的初始化

编程语言的 `main` 函数是程序逻辑的开始，但是一个程序只有逻辑是无法运行的，程序的开头部分需要有一些在 `main` 函数执行之前执行的函数，结束的时候也需要有一些在程序逻辑执行完之后执行的内容。比如全局变量的初始化和回收。

这是通过在 ELF 文件中添加一个段来实现的：`.ctors`（构造）和`.dtors`（析构）。这两个段里面放的是函数列表，在启动和结束的时候会被顺序调用。至于里面是什么，那就与不同的编程语言相关了，有的二进制甚至可以做到不使用这两个段。在 GCC 的 4.7 版本以后用`.init_array`取代了`.ctors`，但是在大部分生成的二进制中还是会保留`.ctors` section，不过里面是空的。这两者的最大区别是`.ctors` 里面可以直接执行的代码，而`.init_array` 里面是函数数组，其顺序调用执行的逻辑是由加载器完成的。

除了`.ctors` 和`.dtors`，完成同样功能的还有`.init` 和`.fini` 这两个段，有的系统两种都支持，有的只支持一种。如果两种都支持，`.init` 会在`.ctors` 执行之前执行。`.init` 和`.fini` 比较老，格式组织也比较混乱，而`.ctors` 和`.dtors` 是单纯的函数顺序调用表，在使用上`.ctors` 系列是`.init` 系列的后继改良版本，但是`.init` 系列仍然是 ELF 的标准结构，并且`.ctor` 是由`.init` 调用的。

在语言上，`.ctors` 和`.dtors` 分别对应 C++ 的 `main` 函数执行之前的构造函数和析构函数。例如全局的对象的构造和析构。因此，这里面的内容并不是真正意义上的程序最早执行的入口。`ld` 有多种方法设置进程的入口地址，通常它按以下优先级顺序指定（编号越前，优先级越高）：

- (1) `ld` 命令行的“-e”选项。
- (2) 连接脚本的 `ENTRY (SYMBOL)` 命令。
- (3) 如果定义了 `start` 符号，使用 `start` 符号值。
- (4) 如果存在`.text` section，使用`.text` section 的第一字节的位置值。
- (5) 使用 0 值。

代码如下。

```
.ctors      :
{
    __CTOR_LIST__ = .;
    LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
    * (.ctors)
```



```

LONG(0)
__CTOR_END__ = .;
}

```

符号 `__CTORS_LIST__` 表示全局构造信息的开始处，符号 `__CTORS_END__` 表示全局构造信息的结束处。符号 `__DTORS_LIST__` 表示全局构造信息的开始处，`__DTORS_END__` 表示全局构造信息的结束处。

这两块信息的开始处是一字长的信息，表示该块信息有多少项数据，然后以值为零的一字长数据结束。

一般来说，GNU C++ 在函数 `__main` 内安排全局构造代码的运行，而 `__main` 函数被初始化代码（在 `main` 函数调用之前执行）调用。

### 10.3.6 进程初始化前的加载

GCC 在编译的时候可以有不错过标准库的选项，如 `-nostartfiles`（其实就是不使用 `crt1.o` 和 `start.s`）、`-nodefaultlibs`、`-nostdlib` 等。`--freestanding` 是一个编译选项，可以用来自定义标准库的函数。

然而在进程启动的时候，`ld.so` 加载程序会去寻找 `_start` 调用（不是 `main`），如果不是用 `crt1.o` 来链接，则不会生成 `_start` 调用，程序自然也就无法启动，所以如果不使用 `glibc`，应该手动定义这个函数作为入口，以便 `ld.so` 加载。

实际在 ELF 启动的时候，`ld.so` 要完成程序的加载和堆栈的准备，然后调用 `__libc_start_main` 开始功能上的准备。此函数会调用 ELF 的构造函数（析构 `__libc_csu_fini` 和构造 `__libc_csu_init`）。完成语言级别的初始化。这一步对应 `.ctors` 或者 `.init`。`.ctor` 也叫 `.init_array`，存放的是一个函数列表。`.init` 在启动的时候会逐次调用 `.ctor` 这个数组里的所有函数。

不是所有的进程都需要 `ld.so`（`ld-linux.so`）来做动态链接的。如果在编译的时候指定了 `-static`，就不需要动态加载器。Golang 就是一个默认不使用加载器的语言。

`Ld.so` 的主要工作是由于在编译的时候采取动态链接的编译方法而导致的。所以需要 `ld.so` 加载这些动态链接的库，重新计算地址，完善程序的调用表。

所以 `ld.so` 需要一个库查找路径的办法，这个路径有很多种提供方法，最常用的是 `/etc/ld.so.cache`，这个 `cache` 文件是添加了一个库之后，调用配套的库缓存管理命

令 `ldconfig` 重新搜索计算生成的，并不会自动生成。所以安装一个新库一般要执行一遍这个程序。按照先后顺序，查找库路径包括以下 6 种方式。

- (1) `LD_PRELOAD` 环境变量指定的路径（一般对应 `/etc/ld.so.preload` 文件）。
  - (2) ELF.dynamic 节中 `DT_RPATH` 入口指定的路径，若 `DT_RUNPATH` 入口不存在（ELF 标准中已经废弃，由 `DT_RUNPATH` 取代）。
  - (3) 环境变量 `LD_LIBRARY_PATH` 指定的路径，但如果可执行文件有 `setuid/setgid` 权限，则忽略这个路径；编译时指定 `--library-path` 会覆盖这个路径。
  - (4) ELF.dynamic 节中 `DT_RUNPATH` 入口指定的路径。
  - (5) `ldconfig` 缓存中的路径（一般对应 `/etc/ld.so.cache` 文件），若编译时使用了“`-z nodeflib`”链接选项，则跳过此步骤。
  - (6) `/lib`，然后 `/usr/lib` 路径，若使用了“`-z nodeflib`”链接选项，则跳过此步骤。
- 我们会发现在 32 位的 Linux 系统上，ELF 的代码加载地址是 `0x8048000`，而在 64 位的 Linux 系统上，则是 `0x400000`。这里的加载指的是 `LOAD segment` 的加载。而 `entry point` 是 `.text` 的入口地址，也就是 `_start` 函数的地址，所以这个数比加载地址稍大。例如在 x64 系统中使用 `ls` 命令后显示的结果，如图 10-27 所示。

```

ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF64
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                               Advanced Micro Devices X86-64
  Version:                               0x1
  Entry point address:                   0x4049a0
  Start of program headers:              64 (bytes into file)
  Start of section headers:              124728 (bytes into file)
  Flags:                                  0x0

```

图 10-27

入口地址距离加载地址的偏移是 `0x49a0` (`entry point - 0x400000`)。而再用 `readelf -S /bin/ls` 查看 section 的情况时，就会发现 `0x409a0` 正好位于 `.text` section 内。`.text` section 地址，如图 10-28 所示。

```

0000000000000000 0000000000000000 AX 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[14] .text PROGBITS 000000000000402a00 00002a00
00000000000011289 0000000000000000 AX 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[15] .fini PROGBITS 000000000000413c8c 00013c8c
0000000000000009 0000000000000000 AX 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[16] .rodata PROGBITS 000000000000413c8c 00013c8c

```

图 10-28

这个默认的地址也是可以通过链接使用“`ld -e -Ttext`”选项来改变的，不过一般

没有这个需要。

x64 内核只会把 LOAD segment 的内容加载到内存中,但是在 x86 的 32 位时代,内核把整个 ELF 头部区域都加载到了内存中,导致直接读取内存就可以获得完整的内存符号表,一些内存注入的算法都是由此产生的。但是在 x64 时代仍然可以直接读取二进制,即使二进制被删除了,也可以在 proc 文件系统下对应 pid 的 exe 文件中找到原始的二进制文件,分析这个文件也能得出各个段在内存中的实际偏移,从而进行注入或修改。

### 10.3.7 链接环境变量

**Ld.so:** 在执行的时候很多环境变量可以用来控制其执行的方法。

**LD\_ASSUME\_KERNEL:** 可以限定内核的版本,如果低于这个要求,ld.so 就拒绝执行。

**LD\_BIND\_NOW:** 可以立刻绑定,而不是延迟绑定。

**LD\_LIBRARY\_PATH:** 可以在编译的时候指定去哪里搜索库,ld.so 就按照这个搜索 (-rpath)。

**LD\_PRELOAD:** 指定在程序执行之前首先加载的库,并且在加载其他库之前先解析。所以这个宏非常不安全。

**LD\_AUDIT:** 可以在程序运行时调用外部的 audit 接口。它也是不安全的。

**LD\_BIND\_NOT:** GOT 和 PLT 表解析后就不更新了,下次调用还解析。

**LD\_DEBUG:** 在有这个宏的时候,ld.so 在执行程序的时候要打印调试信息。这个调试信息是链接器的,打印到 LD\_DEBUG\_OUTPUT 宏指定的位置。

**LD\_DYNAMIC\_WEAK:** 这个允许程序中定义的 weak 符号使用 glibc 中定义的本版本。

**LD\_POINTER\_GUARD:** 安全上的增强,用于管理指针。

**LD\_PROFILE:** 定义哪个共享对象被统计并写到文件 LD\_PROFILE\_OUTPUT 中。

**LD\_SHOW\_AUXV:** auxv 是内核在执行 ELF 文件的时候传输给用户空间的信息。这个变量可以用来看这些值。

**LD\_PREFER\_MAP\_32BIT\_EXEC:** 性能的标志。Mmap 使用的,由于 64 位的机器分值预测错误会影响性能,这个标志会优先使用 32 位的分支预测方法。



### 10.3.8 内核加载 ELF

内核在加载 ELF 文件的时候调用的是 `load_elf_binary()`，首先关心的是 `PT_LOAD` 段，这个段描述了 ELF 的哪些段要被加载到内存。然后是 `bss` 段，将指定大小的内存设置为 0 作为初始化为 0 的 `bss` 段运行时的内存，然后看 `PT_INTERP`，这个段指明了要使用什么加载器来加载这个 ELF 文件的动态库，不指定默认系统自带的（`ld-linux.so.2` 或者 `ld.so` 等）。然后检查 `PT_GNU_STACK`，根据这个值来判断生成的栈是否可以被执行。最后是内核加载加载器，后面的控制就交给用户空间的加载器了。

Linux 内核在加载 ELF 程序时，还会给 ELF 隐式地传递参数，实际上这个参数是传给加载器的。这个参数叫作 ELF 辅助向量（AUXV）。这个向量在大部分情况下看不到，加载器有一个控制变量来控制是否打印 AUXV，例如执行 `$ LD_SHOW_AUXV=1 whoami`，就会打印 `whoami` 的 AUXV，控制变量就是前面的 `LD_SHOW_AUXV`。加载器加载程序的栈结构，如图 10-29 所示。调用 `whoami` 命令时打印辅助向量，图 10-30 所示。

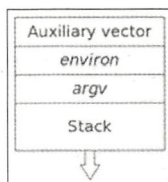


图 10-29

```
root@ubuntu:~/sdb1/yunweishi# LD_SHOW_AUXV=1 whoami
AT_SYSINFO_EHDR: 0x7ffdf62ed000
AT_HWCAP: fabfbff
AT_PAGESZ: 4096
AT_CLKTCK: 100
AT_PHDR: 0x400040
AT_PMENT: 56
AT_PHNUM: 9
AT_BASE: 0x7fc64c9ba000
AT_FLAGS: 0x0
AT_ENTRY: 0x401610
AT_UID: 0
AT_EUID: 0
AT_GID: 0
AT_EGID: 0
AT_SECURE: 0
AT_RANDOM: 0x7ffdf621fae9
AT_EXECPN: /usr/bin/whoami
AT_PLATFORM: x86_64
root
```

图 10-30

这些 AUXV 即使不依赖于加载器的这个支持功能也能在程序内部访问，方法是使用 main 函数的第三个参数，代码如下。

```
#include <stdio.h>
#include <elf.h>
Int main(int argc, char* argv[], char* envp[])
{
    Elf32_auxv_t *auxv;
    while(*envp++ != NULL);
    for (auxv = (Elf32_auxv_t *)envp; auxv->a_type != AT_NULL; auxv++)
    {
        if( auxv->a_type == AT_SYSINFO)
            printf("AT_SYSINFO is: 0x%x\n", auxv->a_un.a_val);
    }
}
```

通过上述代码，就能理解 AUXV 在进程执行中的传递了。

### 10.3.9 Audit 接口

编译的时候可以给 ld 传递 --audit AUDITLIB 参数，如此就会创建一个 DT\_AUDIT section，ld.so 看到这个 section 就会执行 glibc 规定的 audit 接口。在特定的事件发生时，会编译指定库中找到的指定函数来执行。例如当程序调用 dlopen 打开了一个动态库时，就会发生一个事件，从而调用指定库中的 la\_objopen() 函数。

这个接口是用来做连接维度的统计的，更多的时候是被连接器的开发者使用。但是开发者可以利用单独分发的这个库做函数实现版本的管理和替换。例如加载一个库的时候首先调用了 audit 接口，在 audit 接口中，我们将要打开的库替换。

### 10.3.10 一个简单的 ELF 解析程序

一个简单的 ELF 解析程序，代码如下。

```
#include <stdio.h>
#include <string.h>
```

```
#include <errno.h>
#include <elf.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <stdint.h>
#include <sys/stat.h>
#include <fcntl.h>

void parse(char* file_name){
    int fd;
    if ((fd = open(file_name, O_RDONLY)) < 0) {
        perror("open");
        exit(-1);
    }
    struct stat st;
    if (fstat(fd, &st) < 0) {
        perror("fstat");
        exit(-1);
    }
    uint8_t* mem = (uint8_t*)mmap(NULL, st.st_size, PROT_READ,
MAP_PRIVATE, fd, 0);
    if (mem == MAP_FAILED) {
        perror("mmap");
        exit(-1);
    }
    Elf64_Ehdr *ehdr = (Elf64_Ehdr*)mem;
    Elf64_Phdr *phdr = (Elf64_Phdr*)&(mem[ehdr->e_phoff]);
    Elf64_Shdr *shdr = (Elf64_Shdr*)&(mem[ehdr->e_shoff]);
    if (mem[0] != 0x7f && strcmp((const char*)&mem[1], "ELF")) {
        fprintf(stderr, "%s is not an ELF file\n", file_name);
        exit(-1);
    }
    if (ehdr->e_type != ET_EXEC) {
        fprintf(stderr, "%s is not an executable\n", file_name);
        exit(-1);
    }
}
```



```

    printf("Program Entry point: 0x%x\n", ehdr->e_entry);
    char* stringTable =
(char*)&(mem[shdr[ehdr->e_shstrndx].sh_offset]);
    printf("Section header list:%d\n\n", ehdr->e_shnum);
    for (int i = 1; i < ehdr->e_shnum; i++){
        printf("name idx:%d\n", shdr[i].sh_name);
        printf("%s: 0x%x\n", &stringTable[shdr[i].sh_name],
shdr[i].sh_addr);
    }
    printf("\nProgram header list\n\n");
    for (int i = 0; i < ehdr->e_phnum; i++) {
        switch(phdr[i].p_type) {
            case PT_LOAD:
                if (phdr[i].p_offset == 0)
                    printf("Text segment: 0x%x\n",
phdr[i].p_vaddr);
                else
                    printf("Data segment: 0x%x\n",
phdr[i].p_vaddr);
                break;
            case PT_INTERP:
                {
                    char* interp = strdup((char
*)&mem[phdr[i].p_offset]);
                    printf("Interpreter: %s\n", interp);
                    break;
                }
            case PT_NOTE:
                printf("Note segment: 0x%x\n", phdr[i].p_vaddr);
                break;
            case PT_DYNAMIC:
                printf("Dynamic segment: 0x%x\n",
phdr[i].p_vaddr);
                break;
            case PT_PHDR:
                printf("Phdr segment: 0x%x\n", phdr[i].p_vaddr);
                break;

```

```

    }
}

int main(){
    parse("/bin/ls");
}

```

以上是一个修改自 Ryan 的完整可运行的 ELF 解析代码。仔细阅读每一行代码，能够有更加深刻的宏观了解。

## 10.4 ELF的安全性

当有了 root 权限之后，内核就没有秘密了。大部分人即使获得了 root 权限，能看到的東西也不多，其实 Linux 已经提供了所有的信息访问能力，只是大家没有找到查看的方法。

例如我们可以查看任何物理内存的内容，方法是通过打开/dev/mem 设备，然后 mmap 到你的程序，直接读取就可以了。也可以查看任何的内核数据（不只是 proc 和 sys 文件系统暴露的信息），方法是打开/proc/kmem 设备，然后直接读取。

每个进程都可以查看自己的所有可读内存，方法是使用/proc/<pid>/mem，你可能 cat 这个文件永远是错误的，因为不是所有的内存都被进程映射了，尤其是文件开始的位置，所以需要根据/proc/<pid>/mmaps 文件找到具体的文件映射的模式（而如果使用 grsecurity，这个文件就是空），然后 seek 到对应的偏移才能读。这种查看自己内存的需求基本没有，因为既然是我们自己的进程，那么在程序内部自然也就完全可以读取了。从外部查看进程内存的代码如下。

```

#!/usr/bin/env python
import re
maps_file = open("/proc/self/maps", 'r')
mem_file = open("/proc/self/mem", 'r', 0)
for line in maps_file.readlines(): # for each mapped region
    m = re.match(r'([0-9A-Fa-f]+)-([0-9A-Fa-f]+) ([-r])', line)
    if m.group(3) == 'r': # if this is a readable region
        start = int(m.group(1), 16)
        end = int(m.group(2), 16)
        mem_file.seek(start) # seek to region start

```



```

        chunk = mem_file.read(end - start) # read region contents
        print chunk, # dump contents to standard output
maps_file.close()
mem_file.close()

```

如 Gilles 写的上述 Python 代码的读取方式，就可以读取到实际的内存内容。

由于 `/proc/<pid>/mem` 的权限是只有自己可读，所以其他进程如果想读取进程的内存信息就必须使用 `ptrace` 到这个进程。`root` 是可以读到所有信息的，但是程序仍然要暂停才能让外部程序读内存因为直接读内存会导致竞态，可以使用 `gcore` 命令稳定地将整个内存导出到文件中。

执行 `gcore 105676`，得到 `core.105676` 文件，使用 `readelf` 命令查看 `core` 文件，发现全部是与 `mmaps` 对应的内存数据，如图 10-31 所示。

```

root@ubuntu:~# readelf -l core.105676
Elf file type is CORE (Core file)
Entry point 0x0
There are 55 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz              Flags   Align
NOTE           0x000000000000c48 0x000000000000000 0x000000000000000
               0x00000000000130c 0x000000000000000 R       1
LOAD           0x0000000000001f54 0x000055c99f62900 0x000000000000000
               0x000000000000c000 0x000000000000c000 R       1
LOAD           0x000000000000df54 0x000055c99f63500 0x000000000000000
               0x0000000000017000 0x0000000000017000 RW      1
LOAD           0x0000000000024f54 0x000055c99f64c00 0x000000000000000
               0x000000000000c000 0x000000000000c000 RW      1
LOAD           0x0000000000030f54 0x000055c99f74c00 0x000000000000000
               0x000000000002be000 0x000000000002be000 RW      1
LOAD           0x000000000002eef54 0x00007f8dfcc3d00 0x000000000000000
               0x0000000000001000 0x0000000000001000 R       1
LOAD           0x000000000002eff54 0x00007f8dfcc3e00 0x000000000000000
               0x0000000000001000 0x0000000000001000 RW      1
LOAD           0x000000000002f0f54 0x00007f8dfcc3f00 0x000000000000000
               0x0000000000006000 0x0000000000006000 RW      1
LOAD           0x000000000002f6f54 0x00007f8dfce4f00 0x000000000000000
               0x0000000000001000 0x0000000000001000 R       1
LOAD           0x000000000002f7f54 0x00007f8dfce5000 0x000000000000000
               0x0000000000001000 0x0000000000001000 RW      1
LOAD           0x000000000002f8f54 0x00007f8dfd06600 0x000000000000000
               0x0000000000001000 0x0000000000001000 R       1

```

图 10-31

内核后面增加的 `CONFIG_STRICT_DEVMEM` 和 `CONFIG_IO_STRICT_DEVMEM` 等特性逐渐对 `/dev/mem` 文件的访问内存能力进行限制，所以新版的内核已经不是那么容易访问内存的了。

### 10.4.1 二进制修改

工欲善其事必先利其器，二进制修改方面也有专门的工具。最知名的两个是 `elfsh`



和 bap。elfsh 使用 eresi 这个更大的平台，这个平台能做的事情更多，但是已经停止更新了，并且很多组件对 64 位系统的支持也不好。bap 的思路是将二进制转变为中间形式，然后修改再转变回去。

elfsh 实现了一些知名的 ELF 修改手法。例如直接修改动态链接库 .dynamic section 的内容，让动态链接到自己的恶意库。修改 GOT section 中的条目，让符号解析的时候解析到我们定义的符号，而我们自己定义的符号可以直接放在头部的 .interp 的 section 之前或者 .bss section 之后，然后修改 GOT 让其引用到这里。用这种方法把自己的整个代码直接添加到可执行二进制中。

patchelf 工具可以更方便地修改 rpath 和 interpreter，但是 pathcelf 修改 rpath 的方式经常在 libm.so 上是无效的，需要 GCC 编译的时候使用 “-Wl,-rpath” 选项。最强大的二进制修改工具是编程语言，直接写逻辑解析 ELF 格式，并且可以完成任何事情，代码量也不大，但对技能的要求相对较高。

二进制修改技术普遍被用于二进制攻击和二进制加固领域。

## 1. 二进制分析工具

- 常用的二进制分析工具包括：BAT (binary Analysis Tool)、BitBlaze、angr、CodeSonar (商用)、bap、execstack、setarch、ftrace、ERESI project (非常全面的二进制分析工具集合，但是维护者似乎已经不再更新了，它们对 x64 系统的兼容性也一般)、BAP、BARF、scanelf (pax-utils 包)、elfutils 工具包、elfkickers 工具包、volatility (内存审查框架程序，可以用来分析进程内存的实际情况，以及一些常用的内存工具，目前更新比较活跃)、ld --verbose (可以查看到详细的二进制的 section 分布)、bvi (二进制编辑命令) 等分析工具，还有如 PIN、valgrind、DynamoRIO 等的动态分析工具。还有一些分析二进制常用的系统文件，比如 /proc/<pid>/maps、/proc/kcore、/boot/System.map、/proc/kallsyms、/proc/iomem 等。
- Extended core file snapshot (ECFS) 相当于 core dump，但不同的是它 hook 了 core dump 的调用，在生成 core 文件之前，修改这个 core 文件，生成更加详细版本并且兼容正常 core 文件的 ecfs core 文件。它甚至可以不用中断进程的执行而产生 core 文件，这个功能 Linux 本身也是支持的，这对于二进制分析的专业人员来说是非常好的工具。

## 2. 二进制加固工具

一般的安全从业者会熟悉一些常见的 ELF 加固工具，最后一般是某一个工具胜出，大家就都用这一个了。但是由于 ELF 安全加固没有那么大的市场需求，所以完美的市场化产品就不容易产生。目前最知名的应该是 UPX，但是它更多的时候是作为一个压缩自解压的定位存在的。还有一些开源的工具，可以用来研究和轻度使用的，例如 DacryFile、Burneye、objobjf、Shiva by Neil Mehta and Shawn、Maya's Veil by Ryan O'Neill。

## 3. 常用的二进制攻击手法

### (1) 内存攻击

可以在一个正在运行的进程的内存中直接用 ptrace 注入代码，也可以用 so 库的形式注入，然后使用 pthread 将恶意程序运行在宿主进程中。需要注入已有的空间，参考 libptrace 里面有注入并且调用函数的 32 位实现，还会创建一个给我们的恶意二进制运行的额外栈。主要是因为 ptrace 拥有查看和修改寄存器的能力，所以就可以自由控制目标进程的 CPU 执行流程。例如还可以使用 PTRACE\_SYSCALL 的 ptrace 能力定位到跟踪进程的系统调用位置，然后修改寄存器，利用 Linux 的 vdso 机制直接进行我们想要的系统调用，从而不用注入任何代码到 .text section 中（这个 section 现在有 PaX 机制禁止注入）。

如果你的进程使用了 socket，我的 shellcode 可以直接复制（dup2）你的 socket 的 fd 到 0 和 1，然后启动一个 shell，这样这个 shell 的输入、输出就是使用的你的 socket，如此就完成文件句柄的利用了。

使用 LD\_PRELOAD 宏。这个宏指定的库可以覆盖默认的库而被优先加载。

通常二进制攻击和内存攻击可以结合起来使用，能够发挥更大的威力。

### (2) 二进制攻击

- 修改 entry point 来指向注入程序的注入位置。
- 将注入程序的位置写入 .init\_array 作为入口，可以不用修改 entry point，plt、got 也都可以用来作为注入点（需要熟悉加载器的动态链接流程）。
- .got.plt 里面的函数地址默认设置都是 0，加载的时候解析到地址后就不会再解析了。而我们可以直接修改二进制，写入固定的偏移地址，从而让其解析到我们自己的代码位置。这种方法非常简单有效。
- 我们知道 .got.plt 中存储的地址是通过动态加载得到的，这个引用的代码位

于 `.rel.plt` section, 我们也可以修改 `.rel.plt` 里面的加载代码, 使其指向我们自己的代码。

- 由于在二进制中定位到某个函数的实际实现代码非常容易 (`objdump` 就能轻松地将一个函数的位置和内容逆向), 所以我们可以直接在二进制中找到某个函数, 然后直接修改函数的内容, 让其指向我们的代码, 执行完毕后再把控制归还。

### (3) 注入位置

- 将恶意二进制注入到两个 `LOAD` 之间, 也就是代码段和数据段之间, 利用 `segment` 之间的空隙 (`Silvio padding`), 调整代码段的 `size`, 使其属于可执行的代码段区域。
- 直接注入第二个 `LOAD` 的数据区域。但是要在 `.bss` 之前注入, 因为 `.bss` 一定是数据区域的最后一个 `section`。但是现在的二进制一般都会有数据域的不可执行的保护。所以这种方法只可以变相利用。例如将比较大的二进制注入到数据空间中, 然后在其他地方注入小程序, 将大的二进制数据在代码里保存为文件, 然后以 `so` 库的形式加载它们到内存就是可执行的代码了。或者直接 `mmap` 一片可执行的内存, 将数据拷贝过去。
- 最有效的思路是直接创建一个 `segment`, 类型为 `LOAD`, 权限设置为可执行。这样自己的代码就可以非常自由地被安排和执行了。或者可以把其他无用的 `segment`, 例如 `PT_NOTE` 转变为 `PT_LOAD`。

## 4. 常用的二进制加固方法

有攻就有防, 对二进制的防御是基于常见的二进制攻击方法来针对性防御的。对于二进制, 像 `Golang` 程序完全去掉了动态加载系统, 这样大部分的二进制攻击方法都会失效。尤其是进行过 `strip` 的 `Golang` 程序, 几乎可以做到不保留任何的信息。对于二进制攻击者, 几乎只能通过 `entry point` 或者 `.init_array` 等常见的注入头部进行注入, 并且几乎是盲注, 因为你不知道程序本身是做什么的。在程序运行时, 如果使用 `ptrace` 接入进来, 也会发现几乎没有任何可以用来调试的符号。这是在语言选择层面的防御。

由于运行时的注入高度依赖于 `ptrace`, 所以阻止 `ptrace` 的使用就是最有效的防御。可行的方法是自己 `ptrace` 自己, 或者截断 `SIGTRAP` 信号以防止被设置断点, 再或者是动态监测谁在 `ptrace` 自己, 发现了就立刻退出程序。



在二进制层面，可以做代码混淆，让代码不可读，尽可能去掉或者混乱符号。

我们平时见到的杀毒软件，与其说是一个技术产品，不如说是一个市场产品。因为他们大多数都基于签名和病毒库，而我们随手写一个注入就不可能存在于它们的病毒库中。现在的杀毒引擎在很多地方有所提高，不再只是基于签名库，例如内存监测和一些常用的入侵技法的入口检测，以及非常强大的 Windows Defender。

还有另外一个维度的加固。就是我们把原来的二进制文件进行加密压缩等操作，然后放到一个封装之后的头部。这种打包的方案非常适合非入侵式的加固。例如我们可以自己实现一个空的 ELF 桩程序，它的 payload 数据域里面没有东西。主程序就从 payload 域里面取出数据，然后进行密码验证、解压缩、解密等操作，然后再执行最后生成的二进制。而另外一个注入程序使用这个桩程序，将压缩加密之后的程序放到其预留的 payload section 中，这样就完成了一个加固程序。大部分的加固框架逃不出这种架构，可能就是采用具体算法的不同。高级一点的加固算法可以把二进制直接解压到当前的进程空间中，然后将执行流程交付给实际的二进制。

## 10.4.2 二进制格式的病毒和木马

大部分人可能对病毒有所误解，因为大部分病毒不过是一个文件修改程序。我们可以随手写一个修改文件的程序，都可能会导致这个文件不可用，然后搜索可用的可执行文件进行逐个修改，这就是所谓的中病毒了。病毒的传染性才是最关键的，被感染的文件在被执行的时候会感染主机其他文件的能力更重要。而在 Linux 系统下，由于 Linux 系统大部分用于服务器，在包管理等统一的规范化管理下，病毒的传染性几乎丧失，所以病毒的危害性对于 Linux 操作系统来说并不大。

而木马才是当前 Linux 系统的大敌，通常情况下木马被使用漏洞批量地分发，使用 Linux 系统的机器很多都是被木马作为了它的运算能力。所以如何防御木马是二进制安全的核心问题。

直接修改运行的内核的内存代码是最难被主机防御系统发现的，其次是修改进程内存或者配合起来屏蔽系统调用的内核模块。相对容易被发现的是修改机器上的二进制，最容易操作的是使用独立的文件互相守护运行。当然，很多复杂的机制设计配合使用可以达到更好的效果，国内有检测木马能力的互联网公司不多，一线的大型互联网公司也几乎很少有能够检测到进程内存级别的。但是不是防御者不能做

到，而大多数都是因为极少有高水平的攻击者。

### 10.4.3 二进制安全特性简介

#### 1. 可执行栈

随着 C 语言的诞生就诞生了栈的概念，栈一般用来传递参数和记录返回地址。所以只要可以在程序中溢出这个栈，就能修改这个返回地址。以前的栈攻击都是直接溢出，然后把可执行程序放到溢出的地方，被直接执行（stack smashing）。后来操作系统做了很多增强，例如不允许在栈上执行程序，甚至只要用户可写的内存都不准执行程序，甚至硬件都开始支持内存页的可执行性属性。

这些防御能力使得攻击者几乎不能把自己的可执行代码放到程序中执行。攻击者通过溢出漏洞控制程序的跳转执行想要执行外部库函数（这些函数都是已经存在的加载库，所以已经被标记了可以执行）来达到自己的目的，这叫作 Return-oriented programming (rop)。x64 时代的参数直接使用寄存器传递，所以由于这种攻击方式不能直接修改寄存器而受限，而攻击者仍然可以找可以修改寄存器的库代码片段，发生 rop，从而修改寄存器。使用这种机制，只需要一个缓存溢出就可以做任何调用，甚至让操作系统关闭，全局删除数据都可以。

针对 stack smashing 防御方法比较知名的有 DEP 和 Intel 的 Control-Flow Enforcement (CFE)。其实就是 GCC 中实现的金丝雀 (stack canary)，即在真实的栈后面加一片影子空间。如果影子空间被修改了，就说明有溢出发生，但是如果溢出长度超过影子长度就没办法了。只是 Intel 是指令级别的实现，GCC 是函数级别的实现。同样的思路被用在 pthread 的线程 guard 之间。pthread 防止不同的线程之间的栈溢出，就设计了在不同线程的栈之间设置 guard 的机制，防止一个线程数据破坏另外一个线程。同样，超过 guard 长度的溢出可以突破这种防御。

最早的 ELF 攻击把 shellcode 放到栈上，也正是因为如此，这种方法最先被防御。现在一般的栈都没有可执行权限，但是控制这个可执行权限的是 ELF 文件本身的 section，所以如果有 ELF 文件的修改权限这也就不是问题了。用 GCC 编译，可以用“-z execstack”选项打开栈的执行权限，或者使用 execstack 的 shell 命令。

GCC 对 C 语言支持一个扩展，就是函数的嵌套定义。而这个定义是通过将嵌套的函数代码放在栈中执行的，这就要求这种栈是有可执行权限的。而如果代码中没

有使用这个功能，栈的可执行权限就会开启，所以如果代码对安全性要求比较高，就不要使用嵌套函数。在默认情况下，编译器都是在栈对应的 `section` 上，例如在 `GNU_STACK` 上关闭可执行权限，如果对于这个文件有修改的权限就能打破这个封锁。

## 2. return to libc

`return-to-libc` 攻击是一种电脑安全攻击，这种攻击方式一般应用于缓冲区溢出中，其堆栈中的返回地址被替换为另一条指令的地址，并且堆栈的一部分被覆盖以提供其参数。这允许攻击者调用现有函数而无须注入恶意代码到程序中。`libc` 的共享库提供了类 UNIX 操作系统中的 C 语言程序运行时支持，尽管攻击者可以让代码返回到任意位置，但绝大多数情况下的目标都是 `libc`。这是因为 `libc` 总是会被链接到程序中，并且它提供了对攻击者而言一些相当有用的函数（如 `system()` 调用只附加一个参数就可以执行外部程序了）。返回地址可以指向另一个完全不同的区域，这种攻击被称为 `return-to-libc`。

在内存中地址是随机的，所以一般需要首先探测这个地址，探测的方法是使用 `ld.so` 这个动态加载器。因为程序要执行就必须装载并解析动态库，而这个工作就是由 `ld.so` 完成的，所以这是一个现成的入口。

通常这种攻击方法是在栈上执行代码不可用时才会使用的，现在一般的机器都在栈对应的 `section` 上设置了 `NX bit`，这个位可以防止栈数据被执行，可以通过将 ELF 文件的 `GNU_STACK` 这个 `section` 的默认 `RW` 属性改为 `RWE`，以使得栈上有执行代码的权限，如果栈上可以执行代码，`return to libc` 就显得多此一举了。

## 3. 不固定位置编译

`ASLR` 可以把动态库加载到随机的内存地址中，这样就可以增加攻击者的调试难度。但是可执行文件自己在大部分情况下却是有固定的开始执行地址的，这就给攻击者提供了方便。但是仍然有办法让这个地址随机，就是用 `PIE`（`Position Independent Executable`）机制，它可以把二进制编译成与位置无关的文件，而是由内核来完成这个位置无关的随机化过程，所以这个特性需要内核支持。还有一个需求就是要求位置无关的中间文件，就是动态库和 `.o` 这种编译生成的中间代码，这几种所用的技术和思想都是类似的。

如果我们使用 `-fpic` 参数，就可以生成位置无关的动态库，而如果我们使用 `-fPIC`



参数，就可以生成位置无关的可执行文件。这两者在使用上的差别很大，一个是用来给别人加载的；另一个是用来直接执行的。这两者在技术上差别很小，但是它们有两个主要的差别：`-fpic` 生成的文件加上一个 `PT_INTERP` 段和一些启动代码，就比较像 `-fpie` 生成位置无关进程，两者甚至可以用同样的启动代码。由于 `-fpic` 用来生成动态链接库，所以符号不能直接解析到找到的符号，甚至可以允许找不到符号，动态库本身允许引用外部的库，所以在编译自己的时候不需要链接外部的库，只需要把它使用到的外部的库函数放入 PLT 表，链接的时候或者加载的时候解析就可以了。可执行程序要求所有的符号立即解析，并且不允许有解析不了的符号。

ASLR 的主要目的是为了防止 shellcode，它能够让编码在特定位置的 shellcode 无法被找到并执行。要注意的是，ASLR 是一个内核端技术，也就是堆栈等的内存乱序是由内核完成的。Linux 内核默认开启这个技术，`echo 0 > /proc/sys/kernel/randomize_va_space` 命令就可以关掉，同样也是这样开启的。但是有很轻松的办法可以不需要提权就能绕过 ASLR，`setarch `uname -m` -R /bin/bash` 这个命令将设置 bash 启动的时候不使用 ASLR。

这个技术还有一个问题，就是一个程序启动时可能栈地址是随机的，但是当这个程序由其他的程序启动时，这个栈的地址就有了规律，例如使用 `Exec1` 接口调用并启动这个程序。

#### 4. RELRO

我们可以发现 ELF 入侵的主要思路是在本来不是代码的地方注入代码。RELRO 这个 segment 的出现就是为了让一部分的区域变成只读的，例如 `.ctors`、`.dtors`、`.jcr` 等 section 都是经常会被放到这个段里面的。这个技术的只读设置存在于用户端，是编译器和加载器共同完成的。

#### 5. x64

到了 x64 时代，System ABI for x86\_64 被大量使用，而这种 ABI 使得前面的大部分攻击手法的难度都提高了很多，以往的寄存器利用或者栈利用都变得没那么容易了。但是没那么容易是相对于 x32 发展了这么多年形成的成熟的技术套件而言的，随着时间的推移，x64 的攻击也会逐渐成熟。没有绝对的防御，只有难度的提高。

# 11

## 第 11 章

### 存储

本章介绍关于存储的内容，重点介绍通用和底层，包括一般情况下单机的存储上层结构。而虚拟化设计的特殊存储系统则在虚拟化部分单独阐述。

#### 11.1 磁盘管理

##### 1. Devicemapper 与 LVM

Devicemapper 是 LVM 所依赖的内核机制，但是随着 btrfs 文件系统的出现，LVM 就显得多余了。但是目前由于大部分系统上仍然是 ext4 文件系统，所以 LVM 还有很多可使用的场景。

Linux 系统和 Windows 系统都经常面临一个问题，那就是磁盘空间的划分不能有效地长时间的使用。动态调整磁盘空间，在 Windows 系统下有优秀的工具，但是非常费时，并且可能需要关机重启。这对个人用户来说不算什么，所以适用于家用。但 Linux 系统不只是家用，企业通常在有分区扩张需求的时候又不希望重启电脑，并且要较快地完成操作。这时 LVM 就诞生了。现在已经有 LVM 2 了。

LVM 的主要思想是不以硬件的 sda 1、sdb 2 等物理分区为划分分区手段，而是允许组织多个物理磁盘到一个分区，将很多磁盘组成卷组（Volume Group），然后在卷组上随意划分逻辑卷组（Logical Volumes）。

一个卷组叫作 VG (Volume Group); 物理磁盘叫作 PV (Physical Volume); 划分的逻辑卷组叫作 LV (Logical Volume)。在一个 PV 上不像以前划分为文件系统, 而是划分为相同大小的存储单元, 称为 PE (Physical Extents)。默认一个 PE 的大小是 4MB。PE 是 LVM 可以寻址的最小单位。逻辑卷 LV 也被划分为可被寻址的基本单位, 称为 LE。在同一个卷组中, LE 的大小和 PE 是相同的, 并且一一对应。

和非 LVM 系统将包含分区信息的元数据保存在位于分区的起始位置的分区表中一样, 逻辑卷以及卷组相关的元数据也是保存在位于物理卷起始处的 VGDA (卷组描述符区域) 中。VGDA (Volume Group Descriptor Area, 卷组描述符区域) 中包括: PV 描述符、VG 描述符、LV 描述符及 PE 描述符。

系统启动 LVM 时激活 VG, 并将 VGDA 加载至内存, 来识别 LV 的实际物理存储位置。当系统进行 I/O 操作时, 就会根据 VGDA 建立的映射机制来访问实际的物理位置。LVM 并不是取代文件系统, 而是文件系统以下的分区手段。在创建完 LVM 之后, 在 LV 上还要像传统分区一样进行文件系统格式化后才能被使用, LVM 划分的代码如下。

```
# pvcreate /dev/sdb1
# pvcreate /dev/sdb2
# pvcreate /dev/sdb3
# vgcreate volume-group1 /dev/sdb1 /dev/sdb2 /dev/sdb3
```

使用上述命令就可以将 3 个物理磁盘合并为一个卷组。

## 2. LVM 的优点与缺点

LVM 的最大优势是可以快照, 快照在传统磁盘是十分困难的, 其只在 VMware 这种虚拟化机制中容易实现。快照同样也是采用写时拷贝, 通过一个写时拷贝表记录新写入和修改的 PE, 而在修改时并不修改原有的 PE, 而是使用新的, 如此就可以回溯到快照了。

LVM 的另一个优势是其具有可伸缩性。可以无须停机就调整分区的大小。这对于 RAID 系统来说十分有用。也正是 LVM 的设计导致了其拥有了缺点, 就是当一个分区的物理 PE 不连续时, 就会造成极大的性能损耗。

除 LVM 外, 还有其他类似的机制, 如 EVMS、DMRAID。但是 LVM 被广泛采用。这个机制不可能只在用户空间完成, 需要内核空间的代码协助, 这部分代码是 device-mapper (dm\_mod 模块)。dm\_mod 模块完成 I/O 请求的转换工作, 与内存管



理一样，本质上是映射。代码位于 `driver/md`。这部分代码是策略与机制分离的生动体现，策略由用户端指定，机制由内核提供。

这个模块建模了 3 个实体，即 `mappeddevice`、映射表和 `target device`。映射表用来记录两个设备的映射。一个是 `mapped device`，其是内核向外提供的设备和逻辑存在的；另一个是 `target device`，所有对逻辑设备的操作最后都会转变为对 `target device` 的操作，其是物理存在的。通用块层的核心数据请求是 `BIO`，而 `BIO` 不能跨越多个物理设备。因此在映射的情况下，一个 `BIO` 会被克隆分割为多个，然后送到各个 `target device`。

### 3. RAID

磁盘阵列的方式用于做数据存储，这是很多中小企业和家庭的首选。其方案成熟、搭建便捷、价格便宜。大型企业一般会使用云存储或者专用存储系统来处理更庞大的数据。RAID 分为 7 种方式，及这 7 种方式的混合方式，例如 RAID3、RAID6。这些组合方式都是稳定性和效率的权衡选择。RAID 相关的内核模块，如图 11-1 所示。



```

root@ubuntu:/lib/modules/4.4.0-66-generic# find . -name "*raid*"
./kernel/cryptd/asynch_tx/asynch RAID6_recov.ko
./kernel/cryptd/asynch_tx/raid6test.ko
./kernel/lib/raid6
./kernel/lib/raid6/raid6_pq.ko
./kernel/drivers/md/dm-raid.ko
./kernel/drivers/md/raid10.ko
./kernel/drivers/md/raid0.ko
./kernel/drivers/md/raid1.ko
./kernel/drivers/md/raid456.ko
./kernel/drivers/scsi/megaraid
./kernel/drivers/scsi/megaraid/megaraid_mm.ko
./kernel/drivers/scsi/megaraid/megaraid_sas.ko
./kernel/drivers/scsi/megaraid/megaraid_mbox.ko
./kernel/drivers/scsi/pmcraid.ko
./kernel/drivers/scsi/megaraid.ko
./kernel/drivers/scsi/aacraid
./kernel/drivers/scsi/aacraid/aacraid.ko
./kernel/drivers/scsi/raid_class.ko
root@ubuntu:/lib/modules/4.4.0-66-generic#

```

图 11-1

RAID 在 Linux 系统中使用时需要使用对应的内核模块，例如 `RAID10.ko` 是 `RAID0` 和 `RAID1` 的组合 RAID，我们需要什么类型的 RAID 就加载什么模块。

```

modprobe raid0
apt install mdadm
mdadm --create --verbose /dev/md0 --level=stripe --raid-devices=2
/dev/sdb2 /dev/sdc3

```

使用上述命令就能创建 RAID0 的磁盘 md 0, 此后 md 0 就是一个单独的磁盘了, 而这个磁盘物理上包括 sdb 2 和 sdc 3 两个磁盘。

## 11.2 存储协议

### 1. SCSI、ATA 与 T10 的 SAS

1981 年 SCSI 诞生, 其后与 ATA 并行竞争发展。例如串行的 SATA 的出现就导致了串行的 SCSI (SAS) 的出现。SCSI 目前的最新标准是 T10, 而 RAID 则是一种组织多个磁盘的冗余备份或加速的高层次并行架构, ATA 和 SCSI 都支持 RAID。ATA 和 SCSI 关注的主要是如何扩展磁盘和如何传输数据。

由于长期的竞争和互相学习, ATA 与 SCSI 趋同。物理上的不同不在本文的关注点内, 但上层的命令全部使用 SCSI 的命令定义。

在 Linux 系统中会经常见到 SCSI, 大部分 Linux 用户可能只是使用了 SCSI 的命令。一些 SCSI 的背景应该普及一下。1986 年 SCSI 总线标准化, 定义了 8 位并行的总线协议 SCSI-1; 1994 年推出了支持 16 位数据总线的 SCSI-2; 而 SCSI-3 不像之前的协议重点在定义物理层, SCSI-3 是一个文档集 (现代的标准都倾向于用文档集了), 其中有一个就是 SCSI 的命令集, 后来这个命令集单独发展, 被其他物理标准 (如 ATA) 广泛用来作为命令标准 (T10)。

ATA 本来是试图移植高端昂贵的 SCSI 协议为一个便宜简单的协议版本而诞生的, 诞生后独立发展。1992 年推出了 ATA-2; 1995 年又推出 ATA-3。后来又有了 ATA-4、ATA-5、ATA-6, 这些标准一直在升级, 无论是在扩展支持磁盘的容量方面, 还是在提高数据传输的速度方面, 有的标准还增加了一些监控诊断等功能。

在并行时代, SCSI 性能上比 ATA 强, 但进入串行时代, ATA 的优势凸显, 结果是在物理层 ATA 胜出, 在命令集上 SCSI 胜出。于是一个使用 ATA 物理层, 使用 SCSI 命令层的标准就诞生了: T10 委员会制定的 SAS (Serial Attached SCSI)。自此 SCSI 退出了历史的舞台。

SAS 于 2002 年诞生, 与 PCI、USB 总线一样, SAS 也抛弃了总线架构, 开始使用交换架构。还有一个问题, 就是虽然 SCSI 完全退出, SAS 使用了 SATA 的物理层,

但是 SATA 并没有完全合并于 SAS 中，它仍独立发展，SATA 的物理层仍与 SAS 继续竞争（已有 SATA II 和 SATA III），ATA 的命令也在独立发展，然而目前大部分 ATA 设备也都支持 SCSI 的命令。目前的市场情况是 SATA 由于性价比的优势占据了低端市场，SAS 由于高性能的优势占据了高端市场。我们购买的普通 PC 机一般留有 SATA 接口，只可以插 SATA 硬盘。而如果主板上还有 SAS 接口，则既可以插 SATA 硬盘，也可以插 SAS 硬盘。主板之所以普遍不集成 SAS 接口的原因是因为成本太高。SATA 完全可以满足个人用户。

## 2. SCSI 命令族

SCSI 命令包括基本命令和设备相关的命令。基本命令是所有 SCSI 设备都应提供的，术语叫作 SCSI Primary Commands (SPC)。最新的 SPC 版本是 SPC-4，广泛使用的 SPC-3 是第 23 次修改的版本。

SCSI 命令集定义了一个命令模型，客户端（PC）发送一个 SCSI 命令给服务端（磁盘），数据格式是固定的，叫作 CDB，里面有命令类型和参数。磁盘根据命令的执行情况设置 sense key 和 sense code，然而这两个返回值并不会返回到 PC，而需要 PC 再发一条指令去获取才会返回。这样设计本身是为了加速传输，然而 Linux 在实现 USB 驱动的时候执行了一条命令就去获取一次结果，这个机制反而成了累赘（Linux 中有很多这样的做法，因为商业版本的模块提供了更快更好的驱动，例如 ntfs、电梯算法等）。

虽然 ATA 有自己的命令族，但由于 ATA 设备兼容 SCSI 命令族，所以 Linux 内部全部以 SCSI 命令发送。

## 3. 命令的解析

operation code 是每个 CDB 命令的第一个字节，这个字节内部的前三位是 group code，最后 5 位是 command code。所有 CDB 命令的第一个字节是一样的。该字节分为两部分，第一部分 group code 可以用来表示本命令的总长度，有 6 字节、10 字节、12 字节、16 字节这四种长度。第二部分表示具体的命令，有的命令有子命令，处理程序根据 command code 就可以找到与之对应的命令格式来进行解析了。

大部分的命令都有其作用的地址，其地址在 CDB 中的表示形式为一个序号，表



示一个分区的逻辑块的序号，从 0 开始到分区的最大值，如图 11-2 所示。

Table 9. OPERATION CODE byte								
Bit	7	6	5	4	3	2	1	0
	GROUP CODE			COMMAND CODE				

The value in the GROUP CODE field specifies one of the groups shown in Table 10.

Table 10. Group Code values		
Group Code	Meaning	Typical CDB Format
000b	6 byte commands	see Table 2
001b	10 byte commands	see Table 3
010b	10 byte commands	see Table 3
011b	Reserved <sup>a</sup>	
100b	16 byte commands	see Table 5 and Table 6
101b	12 byte commands	see Table 4
110b	Vendor Specific	
111b	Vendor Specific	

图 11-2

#### 4. sense

sense 这个词是 SCSI 命令定义的，当执行一个 SCSI 命令遇到错误时，调用 REQUEST SENSE 给 SCSI 设备，设备就会返回 sense 数据，sense 数据包含上一次具体的出错原因。

当然，凡是标准定义的出错原因，肯定都预定义好了。出错原因只能是预定义原因中的其中一种。由于所有 SCSI 设备共享这个 sense 数据，所以这个“错误”必须能够涵盖各种设备，这也就决定了这个 sense 数据是分层次的。

如图 11-3 所示是 sense 的数据格式，第一行是与所有其他回复命令共享的，因为要辨别回复的种类。对于 sense 数据，RESPONSE CODE 固定为 72h 或 73h。

SENSE KEY 是第一层大类。描述得比较宏观，得不到具体的信息。例如 NO SENSE 表示没问题；RECOVERED ERROR 表示命令已经正常执行，但带来一些错误，这些错误也是可以恢复的；NOT READY 表示设备还未准备就绪；MEDIUM ERROR 表示传输和存储媒体上的错误，等等，共有 14 种错误类型。

Bit Byte	7	6	5	4	3	2	1	0
0	Reserved	RESPONSE CODE (72h or 73h)						
1	Reserved				SENSE KEY			
2	ADDITIONAL SENSE CODE							
3	ADDITIONAL SENSE CODE QUALIFIER							
4	Reserved							
5								
6								
7	ADDITIONAL SENSE LENGTH (n-7)							
Sense Data Descriptor(s)								
8	SENSE DATA DESCRIPTOR 0 (see table 14)							
n	SENSE DATA DESCRIPTOR X (see table 14)							

图 11-3

## 5. 用户端直接调用 SCSI 命令

我们知道在内核中对 SCSI 命令的转换发生在 SCSI 层。而 SCSI 分为 3 层，最上层是针对不同设备的。sd 表示磁盘；sr 表示光盘；st 表示磁带；sg 表示通用。我们要关注的就是通用。文件系统向下调用磁盘中的文件需要用到的是 sd，而 sg 内核驱动的存在使我们可以不使用文件系统，直接在用户空间调用 SCSI 命令。

你可以使用 sg 模块暴露出来的 API 来发送 SCSI 命令，也可以使用已有的程序集。这个程序集是 sg3\_utils。你可以用 apt 或者 yum 命令安装这个包。安装之后键入 sg，就会发现有很多 sg 开头的程序。例如 sg\_inq /dev/sg0 命令查询 sg0 的信息，通常可作为硬件的 ping。sg\_raw 命令可以发送用户自己定义的任意格式的软件。

## 6. SCSI (Small Computer System Interface) 层

SCSI 协议可分为 SCSI-1 (5MB/S)、SCSI-2 (20MB/S)、SCSI-3 (并行多个子版本，传输速度过百兆)，但这些都不是 Linux 系统中所指的 SCSI。Linux 系统中的软件 SCSI 只包含 SCSI 命令，其内容是根据数据请求构造命令，执行命令，反馈命

令执行的结果。

所有的存储设备都使用 SCSI 命令，只是数据传输的方式不一致，因此 Linux 系统通过 SCSI 层访问所有块设备，在内核中 SCSI 层包含上、中、下三层。

上层的作用有 3 个：区分不同的 SCSI 设备；将通用块层的数据请求转变为 SCSI 命令；向通用块层返回其数据请求的执行结果。

中层的作用有 5 个：抽象下层不同总线的操作为统一的 API；提供注册和管理多个不同种类型下层总线设备；为下层的设备提供错误和超时处理能力；将来自上层的 SCSI 命令进行排列并维护该队列；向较高层报告其 SCSI 命令执行的结果。这一层次与 SCSI 的 specification 直接相关，可以说是对 SCSI 总线的驱动实现。

下层的作用是唯一的，即定义针对各种不同的 SCSI 适配器的操作接口，但是都对应到向上注册映射到中间层的标准 API 上。

由于磁盘是接收 SCSI 命令的，所以通用块层并不知道 SCSI 命令的存在。

#### (1) 下层

下层的 SCSI 适配器的种类较多，但是在 PC 中一般是插在 PCI-E 上的 SCSI host 适配器（也叫作总线控制器），该 host 为物理上 SCSI 总线的起点，但却是内核中 SCSI 驱动的终点，一般是一个 SCSI 总线控制芯片。每个这种 host 可以支持多个 channel，每个 channel 上可以连接多个 SCSI 节点（node），每个节点可能包括多个设备，这些挂载在 SCSI 总线上的设备称为 LUN（Logical Unit）。

但是在 Linux 的 SCSI 子系统中，SCSI Host 不但可以指 SCSI 标准中规定的 SCSI 总线；还有可能是指一个虚拟的 SCSI 总线设备，任何人都可以实现 SCSI Host 所需要的函数操作，从而定义一个符合自己要求的 SCSI 虚拟设备；也有可能是一个 USB 总线控制器，这样所有通过 USB 插口接入本机的设备只需要实现一个 SCSI Host 对应操作，即可挂载到内核的接触范围内；也可以是一个 IDE 总线控制器，这样所有在该总线上的设备也可以挂载在 SCSI 子系统中了。由于现在的硬件 PCI 总线一般作为主要总线存在，类似 USB 总线控制器、SCSI 总线控制器等，这些附加总线首先是作为一个 PCI 设备挂载到 PCI 总线上的，这种作为 PCI 设备的总线控制器叫作 HBA（主机总线适配器）。

内核驱动代码也必须对未来有可能接入系统的各种设备进行定义。对于 LUN 的定义是位于中间层的 `scsi_device` 结构体。而对于 node 的定义是中间层的 `scsi_target` 结构体，channel 没有对应的结构体。

系统中也有可能同时存在多个 SCSI 控制芯片，即多个 SCSI host。对于如何定



位每个 LUN 设备就需要一种编码方式。根据拓扑结构可以很容易地知道定位的编码方式是：`host_id`、`channel_id`、`node_id`、`lun_id`。这些 ID 的生成方式不讨论，但是根据每个设备的编号就可以定位到具体的单个 LUN 设备了。

## (2) 中层

从下层的描述可以知道，只有 `scsi_host` 以及对应物理上的 SCSI 总线控制器属于下层，而 `scsi_target` 和 `scsi_device` 则属于中层的数据结构，并且中层也定义了 `scsi_host` 的标准操作（`scsi_host_template` 结构体）。该结构体的具体函数内容由下层定义。

`scsi_host_template` 中有一个很重要的函数，即 `queuecommand`。该函数可以对应不同 `scsi_host` 中不同的操作。其意义都是将上层提交的命令加入命令队列进行处理。这个队列的深度可以只有 1，也可以支持多个命令的排列。这是不同的 `scsi_host` 的实现而各自决定的。由于 `scsi_target` 和 `scsi_device` 在本层实现，这里还需要根据命令的执行情况动态更新所存储的对应设备的信息。

这里要特别注意的是，`scsi_target` 和 `scsi_device` 并不是存在本机的物理实体，而是外设在本机内存中的建模，表示了外设的状况。

## (3) 上层

较高层区分不同的 SCSI 设备类型，典型的类型包括磁盘（`sd`）、磁带（`st`）、CD（`sr`），还有一个通用设备（`sg`），分别定义了这 4 个驱动，括号内是内核对其的简称。不同的设备可以将针对不同设备的特定请求转化为对应的 SCSI 命令。通用设备（`sg`）不对应任何具体的设备类型，用户端可以直接使用 `sg` 提供的接口向任何支持 SCSI 命令的设备发送 SCSI 命令，典型的用户端支持程序是 `sg3utils` 包。

通过对中层的讨论，具备了上层如何将通用块层下传的理论基础。上层从通用块层接收到了数据访问的请求，将其转化为 SCSI 命令，这个命令在上层中定义为 `scsi_cmnd` 结构体。然后调用中层的 `scsi_host_template` 结构体中定义的 `queuecommand` 接口，将此命令交付给中层处理。

在命令处理结束时，本层的回调函数会被以软中断的形式调用，以处理与命令相关的后续操作来通知通用块层该条命令的执行结果。

## 7. SCSI 命令

SCSI 命令有 256 种，每个 SCSI 命令被组织成 CDB 的数据结构后发送给磁盘，每个磁盘都识别 SCSI 命令的 CDB 结构。CDB 结构包含了命令类型和命令参数。包

含如下两个域。

- **Operation Code:** 256 种 SCSI 命令的其中一种。而这个域被进一步拆解为 group code 和 command code, 前面的 group code 表示 CDB 的总长度, 一共有 8 种, 不过预定义的只有 6 字节、10 字节、12 字节、16 字节四种长度的 CDB。
- **Control:** 所有命令都有, 携带标志位, 可厂商定义。  
以下都不是必需的域, 而是不同的命令对应不同的域。
- **Service Action:** 子命令, 并不是所有 CDB 都有, 在某个 Operation Code 下的不同子命令用这个域区分。
- **Logical Block Address:** 磁盘的逻辑地址, 表示命令操作的内容。
- **Transfer Length:** 在传送数据时表示要传送的数据的长度。
- **Parameter list length:** 命令参数的长度。
- **Allocation Length:** 说明客户端可用的接收缓存的大小。

#### (1) response

对 CDB 命令的响应命令叫作 sense。但是这个响应不是自动产生的, 需要 SCSI 设备主动使用 sense request 命令查询。所以对于发送请求方来说, 命令的执行分为两个阶段, 发送成功和磁盘设备执行成功。函数调用结束的状态只表示是本机发送该命令的结果状态, 而不表示实际磁盘设备的执行情况。如果想要获得执行情况, 则必须手动获取 sense 数据。

目前 Linux 系统的 SCSI 实现就是两个阶段的回调: 一个是处理本机处理结果, 另一个是发送 sense request 查询设备的执行结果, 这样才会继续向下执行。

#### (2) scsi\_cmnd

承载 CDB 的是 scsi\_cmnd 结构体, 值得注意的是, CDB 中定义了很多结构域, 但是这些结构域丝毫没有对应到 scsi\_cmnd 结构体中。而在这个结构体中, CDB 仅仅是以一个字符串指针的形式存在的。既然如此, 那这个结构体有什么作用呢? 答案是管理作用。scsi\_cmnd 结构体定义如下。

```
struct scsi_cmnd {
    struct scsi_device *device; //设备指针
    structlist_head list; /*scsi_cmnd是一个列表*/
    structlist_head eh_entry;
    structdelayed_work abort_work; //执行这个命令的工作队列
    int eh_eflags;
```

//唯一定位与追踪, 给每个命令分配一个唯一的 ID: serial\_number, 在命令生成时产生, 执行结束时销毁

```

unsignedlong serial_number;
unsignedlong jiffies_at_alloc; //创建时间
int retries; //命令被重试的次数
int allowed;
unsignedchar prot_op;
unsignedchar prot_type;
unsignedchar prot_flags;

//命令体相关
unsignedshort cmd_len; //命令体长度
enum dma_data_direction sc_data_direction;
unsigned char *cmd; //命令体
struct scsi_data_buffer sdb;
struct scsi_data_buffer *prot_sdb;

unsigned underflow; //如果实际传输的数据少于这个值就会返回错误
unsigned transfersize; //在发生故障或断开连接前保证的最小传输单元

struct request *request; //通用块层的请求指针
unsignedchar *sense_buffer; //命令结果
void(*scsi_done)(struct scsi_cmnd *); //命令执行完的回调函数
struct scsi_pointer SCp;
unsigned char *host_scribble; //host 可以申请内存放在这里, 也可以释放
int result; /* 下层驱动的处理结果 */
unsignedchar tag; /* SCSI-II queued command tag */
};

```

## 8. 数据存储机制

检查 SCSI 的数据完整性有两种机制, 即 DIF 机制和 DIX 机制。DIF 机制需要收到磁盘和访问磁盘的操作系统的两方面支持。其在每个 sector 后面加 8 个字节的保护信息。这样一个 sector 的大小就变成了 520 (原来是 512) 字节。而这 8 个字节的计算是要在 HBA 硬件中完成的, 然后由 HBA 一起传送给磁盘设备。不仅有 DIF 机制, 还有其他种类的检查数据完整性方式, 都是添加额外的数据到 sector, 文件系统通过合理地安排数据, 甚至可以不增加 sector 的大小, 而利用已有的数据空间,



安排一部分出来做完整性信息的存储，但是目前的磁盘都提供了额外的空间，所以文件系统的做法在很大程度上就没必要了，但是想要获得更大的辅助空间，就可以利用数据交织和磁盘提供的有限的额外的位来提供更多的位。额外的位也并不一定用来存储校验信息，可以是 tag，是由文件系统来决定存放什么。

而对于 HBA 不支持自动计算 DIF 的设备，就需要在内核中计算，然后由内核一起传递给磁盘设备。这种内核中计算校验信息的机制叫作 DIX (Data Integrity Extension)。

## 9. 磁盘驱动

磁盘驱动的文件是 sd.c 和 sd.h，抽象的设备结构体是 struct scsi\_disk，而这个设备是更上层的 gendisk 的一种子类型。

sd 设备的读写初始化函数是 sd\_setup\_read\_write\_cmnd(struct scsi\_cmnd \*SCpnt)。我们可以尝试分析并进行优化，下面是函数流程。

- 获得要读写的内存位置和大小。
  - 初始化命令携带的数据存储结构（初始化 scatterlist，将 bio 中携带的上层数据映射到 scatterlist 中）。
  - 检查读写命令是否错误（设备是否在线、读写大小是否超出界限、设备正在发生的变化）。
  - 处理特殊情况（sd 卡不能连续读取最后几个 sector、读写的大小的最低位和设备实际的最小 sector 大小不一致（上层提交下来的全部是 512 字节为单位的读写单元））。
  - 实际生成命令。
    - (1) 读全部使用 READ\_6，写全部使用 WRITE\_6 初始化命令头部。
    - (2) 写数据需要检查数据完整性。
    - (3) DIF、DIX 检查和处理（这是数据完整的特性，可以在数据后存储校验值）。
    - (4) 根据读写大小，重新初始化命令头部为 READ\_10、READ\_12、READ\_16 等。
  - 初始化命令的其他域。
- 通过上面的逻辑可以看出，可以用以下方法提高效率。

- 特殊情况处理（效果欠佳）。
- 去掉 DIF、DIX 检查（效果很好）。

需要注意的是，这里只有初始化读写的函数，没有实际的发送函数，实际的发送函数是由更底层的驱动（如 USB）执行的，并且在很多时候数据通道的瓶颈并不是 CPU 的处理能力。

## 10. 回调函数

在 `sd.c` 中生成命令，命令执行完后发生回调。当然，在此之前底层也会有向下的发送和向上的回调发生，但是这里不考虑。

回调函数是 `sd_done`。此时 `sense` 数据已经由底层的回调获得了，但是对 `sense` 数据进行判断和处理却是在这里执行的。

## 11. `scsi_driver`

实际的执行函数构成了 `struct scsi_driver` 结构体的域。也就是说，这些操作就是这个层次的 SCSI 驱动。驱动结构体的代码如下。

```
static struct scsi_driver sd_template = {
    .owner                = THIS_MODULE,
    .gendrv = {
        .name              = "sd",
        .probe              = sd_probe,
        .remove             = sd_remove,
        .shutdown           = sd_shutdown,
        .pm                 = &sd_pm_ops,
    },
    .rescan                = sd_rescan,
    .init_command           = sd_init_command,
    .uninit_command         = sd_uninit_command,
    .done                   = sd_done,
    .eh_action              = sd_eh_action,
};
```

一个驱动包括设备的检测、电源管理；设备的添加和删除、扫描；设备的使用和错误处理等，与很多其他数据结构类似，驱动就是一个函数指针结构体。

内核中对上层、中层、下层的划分与物理概念是相反的。最上层是最抽象的设备类型；中层是不关心设备类型的通用设备的定义；最下层是不关心设备定义的传输方式接口。

## 11.3 通用块层抽象

### 11.3.1 通用块层功能概览

通用块层位于 SCSI 的上层，文件系统的下层，系统主要的内存管理和读写优化都是在这里完成的。DIRECT\_IO 是跳过这一层的。这一层不是驱动，而是一种机制。其代码位于 linux/block 文件夹内，是单列出来的。

我们先不看代码，分析一下这一层都需要什么组件。

- 对磁盘的抽象 genhd.c 和对分区的抽象：partition-generic.c 和 partitions 目录下的文件。
- 上层文件系统会把对文件的访问转变为对多个 sector 的访问，这些 sector 很可能在内存中是分离的。所以需要一种数据表示方法，用来表示要读写的数据内容。这个数据结构叫作 bio。
- SCSI 标准相关内容。

(1) 新的 SCSI 标准有 DIF 和 DIX 的数据保护机制，无论对于读还是写的数据，都需要一个数据完整性的校验，由于在通用块层存储数据的结构体是 bio，所以对其进行校验的文件叫作 bio-integrity.c。这个文件完成的是与内存相关的设置，真正的算法在 blk-integrity.c 中定义一系列钩子函数。不同的硬件会注册不同的计算方法供本层调用。也就是说，这里实际实现的是 DIX 协议。

(2) 本层要知道 SCSI 的接口，本层定义了 bsg (block SCSI generic device) 的 v4 接口，在 bsg.c 和 bsg-lib.c 文件中。

(3) T10 保护的支持算法位于 t0-pi.c 文件，SCSI 的 ioctl 逻辑位于 scsi\_ioctl.c 文件。

- 连接本层各个功能组件的核心程序：blk-core.c，还包括一些实现特定周边的辅助文件。这一部分包括以下文件。



- (1) blk-core.c: 内核执行这部分代码不是阻塞的, 而是使用内核线程完成的, 使用的是 kblockd, 其定义和相关功能位于 blk-core.c 中。
- (2) bio.c: bio 只是数据的存储结构, 但是一个命令请求不只有数据, 还需要有其他控制和状态信息, 这些信息和 bio 一起被组织到 request 中。但是要注意的是, request 和 bio 都只是本层的数据结构, request 服务于电梯算法, bio 用于存储用户传进内核的数据。
- (3) blk-map.c: 将用户数据映射到 bio 结构体。
- (4) blk-merge.c: 将 request 中的 bio 数据映射到下层 (SCSI) 使用的 scatterlist 结构体的处理程序。
- (5) blk-timeout.c: 请求如果超过了一定的时间需要被 time out。
- (6) bounce.c: 请求到的数据在本层需要有缓冲, 可以从中提取提交到上层所需要的数据, 而丢弃或者缓存一部分上层没有用到的数据。这种行为叫作 bounce。
- (7) 队列 (queue) 处理 (对于块设备的一系列命令, 需要队列缓存, 并且这一层最重要的是队列中的各个命令, 有可能可以合并为一个命令, 例如读取连续数据的两个命令, 由于每次存取数据的量越大, 越节省时间, 所以这一步是提高传输速度的关键)。
- (8) blk-exe.c: Linux 的设计者将对 queue 的插入执行操作单独提取出来放到其中。
- (9) blk-settings.c: 对队列的属性进行设置。
- (10) blk-tag.c: 对队列中的请求添加 ID (tag), 可以通过该 tag 直接找到该请求。
- (11) blk-throttle.c: 凡是通信管道都要考虑流量控制问题。queue 可以有多个来源, 如果某个来源瞬间提交了过多的 bio, 那么其他来源的 bio 就可能饥饿。如果想防止这种现象发生, 就需要给队列针对某一个来源添加一个阈值, 这个阈值由 blk-throttle.c 控制。
- (12) elevator.c: 电梯算法接口。合并多个请求的操作, 需要有合并的算法, 合并的算法有很多, 但是核心部分是要为这些算法提供调用的接口函数。
- (13) 提交请求。当电梯算法被执行完, 多个请求和其对应的 bio 被合并, 这个 bio 就需要被提交到下层 (SCSI 的上层) 去实际地执行发送。发送完

毕后还要执行回调。这部分代码也在这里提供。

- 电梯算法：电梯算法在 `queue` 上执行合并操作，是性能优化的关键。代码位于 `elevator.c`、`deadline-iosched.c`、`cfq-iosched.c`、`noop-iosched.c`，还有提供优先级的 `ioprio.c` 文件。
- 对于 I/O 上下文的处理。I/O 上下文是在请求上层的数据结构，如果说通用块层处理请求级别的数据结构，文件系统就是处理 I/O 上下文的。而文件系统层次包括同步和异步两种数据模式，这里的 I/O 上下文 (`io_context`) 主要用在异步模式，异步 I/O 在提交 I/O 请求前必须要初始化一个 I/O 上下文，一个 I/O 上下文会包含多个请求。通用块层对 I/O 上下文的处理函数放在 `blk-ioc.c` 文件中。
- 正常的逻辑是发送了 I/O 命令，命令请求完毕后会调用回调函数。但是通用块层允许 poll 操作，就是没有回调函数，请求执行完成后需要用户手动查询和处理。这部分代码在 `blk-iopoll.c` 文件中。
- 对本层命令队列的处理可以有一个 CPU，也可以有多个。如果是多个 CPU，就需要对队列进行特殊优化，这种优化叫作 mq。相关代码位于 `blk-mq.c`、`blk-mq-cpu.c`、`blk-mq-cpumap.c`、`blk-mq.h`、`blk-mq-sysfs.c`、`blk-mq-tag.c` 和 `blk-mq-tag.h` 文件中。
- 内核处理命令的返回结果，在通用块层不可能是使用硬中断，所以这里的回调使用的是软中断，定义在 `blk-softirq.c` 文件中。
- 实现 sysfs 接口，定义在 `blk-sysfs.c` 文件中，实现 cgroup 子系统的 `blk-cgroup.c`。
- 其他的辅助功能组件：将内容 flush 进磁盘的 `blk-flush.c`、辅助函数 `blk-lib.c`、用来解析磁盘信息返回值的 `cmdline-parser.c`、提供 ioctl 接口的 `compat_ioctl.c` 和 `ioctl.c` 文件。

从以上内容可以看出，这一部分的关键组件是 `request`、`queue`、`bio`、`elevator` 和磁盘与分区的抽象。

### 11.3.2 数据完整性校验

如果要对 `bio` 进行数据完整性校验，需要调用 `bio_integrity_alloc` 给 `bio` 分配对应的空间，然后通过 `bio_integrity_add_page` 给 `bio` 添加额外的空间，用 `bio_free` 就

会自动删除分配的空间。具体的计算 bip (dif) 的算法由具体的驱动提供，驱动调用的是 `blk_integrity_register` 来注册自己的计算函数。

在文件系统中，可以通过 `/sys/block/<bdev>/integrity/` 目录下的 `write_generate` 和 `read_verify` 来控制是否执行读写校验。在大部分情况下，数据完整性对于文件系统是透明的，但上层的文件系统仍可以显式地使用 DIX 机制。在 `bio_integrity_enabled` 为 1 的情况下，上层调用 `bio_integrity_prep` 为 bio 准备 bip。磁盘设备在注册时可以生成 `blk_integrity` 结构体，体现就是存放具体的读写校验函数和 tag 的大小。

### 11.3.3 设备抽象

我们知道要抽象参与的组件为一个一个的结构体，要传输数据到磁盘，需要磁盘的抽象，而又不只是操作磁盘，所以必然要有一个更通用的抽象。操作需要一个抽象的函数结构体，操作的具体命令也需要一个统一定义的结构体接口。

Linux 的通用块层对磁盘的抽象是 `gendisk` 结构体，该层以下的各种设备都是这个结构体的一种。例如 SCSI 磁盘设备 `scsi_disk` 就是 `gendisk` 的一种。对于分区的抽象是 `struct partition`。设备驱动的抽象是 `block_device_operations` 结构体，对设备进行指令操作的结构体是 `struct request`，连接通用块层和下层设备指令操作的数据结构是 `bio`，`bio` 在请求中既能被上层识别，也能被下层识别。

### 11.3.4 BIO 和 bio\_set

BIO 是通用块层表达数据的方式，其将用户传递进来的数据转换为 bio 存储，bio 又包含了请求。多个 bio 可以组成链接，bio 中提供链表结构。bio 结构体定义如下。

```
struct bio {
    struct bio          *bi_next;      /*BIO 链表*/
    struct block_device *bi_bdev;      //文件系统层的块设备抽象
    unsigned long       bi_flags; /* bio 的状态，例如 BIO_SEG_
VALID。bio_flagged(bio,flag) 用于检测 bio 的 bi_flags 域是否与 flag 相等*/
    unsigned long       bi_rw;        /* 标示是读还是写的标志位 */
}
```



```

        struct bvec_iter      bi_iter;
        unsigned int          bi_phys_segments; //这里有了有效值之后
BIO_SEG_VALID 标志才会被设置
        unsigned int          bi_seg_front_size; //用来计算
segment 大小
        unsigned int          bi_seg_back_size;
        atomic_t              bi_remaining;
        bio_end_io_t          *bi_end_io; //BIO 全部执行结束的回调函数
        void                  *bi_private;
        unsigned short        bi_vcnt;      /*bio_vec 的数目*/
        unsigned short        bi_max_vecs; /*能够持有的最大
bio_vecs 数目*/
        atomic_t              bi_cnt;
        struct bio_vec         *bi_io_vec;   /*实际的数据数组列表*/
        struct bio_set         *bi_pool;
        struct bio_vec         bi_inline_vecs[0]; //用户链接多个 bio
的 0 长度数组
    };

```

内核里有一个 bio 和多个 bio\_vec，一个 bio\_vec 即为一个 segment。由于上层提交 bio 中的 bio\_vec，所以 bio 本身也是可以合并的。每个 queue 可以有各自的标志位，QUEUE\_FLAG\_NO\_SG\_MERGE 控制是否允许 bio 合并。这样 bio 就有了两种统计方式：bi\_vcnt 表示 bio 没有经过自身合并的 bio\_vec 数目；bi\_phys\_segments 表示将物理连续的 bio\_vec 算成一个后统计出来的段总数。这里需要注意的是，bio 的段总数并不是单个 bio 的段的数目，因为 bio 是个链表，所以段的数目总数统计的是链表中段的总数。

### 11.3.5 request

request 中包含了 bio 和其他参数，例如表明携带数据总大小的 \_\_data\_len。用双下画线的域一般是不直接使用的，而是要使用辅助函数调用，典型的获得 \_\_data\_len 的函数接口的是 blk\_rq\_bytes(const struct request \*rq)，blk\_rq\_sectors 可以返回这个 request 携带的 sector 的数目，代码如下。

```
static inline unsigned int blk_rq_sectors(const struct request *rq)
```

```

{
    return blk_rq_bytes(rq) >> 9;
}

```

request 结构体比较大，这里就不列出了。如果说 bio 是数据层面的，request 结构体就是业务层面的。结构体的定义都和功能相关。由于多个 bio 可以被合并到一个 request 中，所以 request 要为这种功能提供支持。bio 合并到 request 中既可以在原 bio 的前面合并，也可以在后面合并。如果在前面合并，那么肯定是在最前面，此时直接利用 bio 本身的链表结构插入到最前面即可。如果在后面合并，也肯定是在最后面，但是此时没有使用 bio 本身的链表结构，而是使用了一个额外的域，让 biotail 来存储要合并进入的 bio。因为这个域本身的定义就是用来放最后一个 bio 的。向前合并最后一个 bio 不变，而向后合并最后一个 bio 要变化。request 中的域分为 3 类，分别用在 3 个不同的地方，即驱动、通用块层、I/O 调度。不仅多个 bio 可以合并到一个 request 中，多个 request 也可以合并为一个 request，这个合并就是通用块层最核心的电梯算法的功能（实际合并的还是 bio）。

在 request 的 flag 中：REQ\_FLUSH 表示执行 bio 前进行 flush；REQ\_FUA 表示执行 bio 后进行 flush；QUEUE\_FLAG\_NO\_SG\_MERGE 表示是否允许 bio 本身的 bio\_vec 进行物理合并。

### 11.3.6 request\_queue

这是通用块层的请求队列，这个队列每个 CPU 有一个。上层的数据请求首先生成 bio，然后由 bio 生成 request，再添加到 request\_queue 里，最后 request\_queue 会被执行。这个执行包括很多步骤，最重要的是电梯算法。每个算法都会在全局的 request\_queue 之外生成自己的队列结构体 elevator\_queue。

request\_queue 中有挂载的电梯算法的队列，并且还有为电梯算法服务的域，例如 last\_merge 表示上次合并的 request。利用这个域相当于使用 cache，可以首先尝试与 last\_merge 指定的 request 进行合并，因为连续数据的概率很大。request\_queue 链表的代码如下。

```

struct request_queue {
    struct list_head    queue_head;

```

```
//之后略
```

```
};
```

这里的第一个元素是 `queue_head`，是 Linux 内核特殊的 list 定义法，这种定义法可以把不同的结构体串成一个 list，list 的第一个元素就是 `request_queue`，后续的都是 `request`。也就是说，后面来的新的 `request` 都是添加到这个队列中的。

队列有很多属性，都是用宏定义的。队列也有一个专门的结构体来定义队列的极限，即 `struct queue_limits`。比如其中的 `unsigned short max_segments` 域表示本队列最多可存放的物理 `segment` 数，在合并操作前要检查合并前队列的总物理段数加上合并的物理段数是否超过这个数。队列的极限和属性对电梯算法非常重要，是电梯算法主要参考和修改的内容。

### 11.3.7 电梯算法

要想实现电梯算法，就需要知道电梯算法相关的元素。

- 每个电梯算法的具体函数作为一个函数表要定义 `struct elevator_type` 结构体。
- 每个电梯算法都要有自己的队列组织（可以有多个队列），结构体是 `struct elevator_queue`。
- 核心元素是 `struct elevator_type` 和 `struct elevator_queue`。定义好以上两个结构后，使用 `elv_register` 注册 `elevator_type`，将 `request_queue` 的 `elevator` 域赋值为定义的 `elevator_queue` 即可。这样系统在处理 `request_queue` 调用电梯算法时就可以找到算法的数据和函数了。

要了解电梯算法的工作原理，具体的算法可以先略过，找到其框架流程更重要。这个流程函数是 `blk_queue_bio(struct request_queue *q, struct bio *bio)`。一个参数是要插入的 `request` 队列，另一个参数是传递下来的 `bio` 数据。当然在这个函数之上，作为整个通用块层的提交请求的入口函数是 `void submit_bio(int rw, struct bio *bio)`。而 `submit_bio` 本质上是做一些统计记录之后就调用 `generic_make_request`。`generic_make_request` 的返回值不是使用函数返回值，而是使用 `bio` 本身提供的回调函数 `bio->bi_end_io`，如图 11-4 所示。



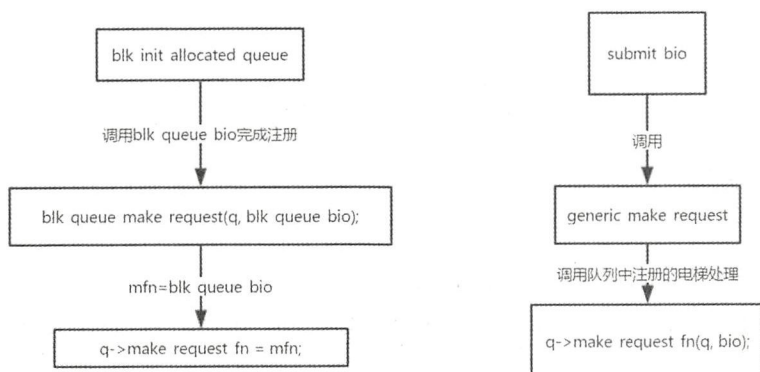


图 11-4

## 1. generic\_make\_request

generic\_make\_request 函数的代码如下。

```

blk_qc_t generic_make_request(struct bio *bio) {
    struct bio_list bio_list_on_stack;
    blk_qc_t ret = BLK_QC_T_NONE;
    if (!generic_make_request_checks(bio))
        goto out;
    if (current->bio_list) {
        bio_list_add(current->bio_list, bio);
        goto out;
    }
    BUG_ON(bio->bi_next);
    bio_list_init(&bio_list_on_stack);
    current->bio_list = &bio_list_on_stack;
    do {
        struct request_queue *q = bdev_get_queue(bio->bi_bdev);
        if (likely(blk_queue_enter(q, false) == 0)) {
            ret = q->make_request_fn(q, bio);
            blk_queue_exit(q);
            bio = bio_list_pop(current->bio_list);
        } else {
            struct bio *bio_next = bio_list_pop(current->bio_list);
            bio_io_error(bio);
            bio = bio_next;
        }
    } while (bio);
    return ret;
}
  
```

```

    }
    } while (bio);
    current->bio_list = NULL; /* deactivate */
out:
    return ret;
}

```

这个函数从 Linux 内核的老版本到最新的已经有比较大的改变，不变的是首先进行 bio 检查，然后调用 `make_request_fn (blk_queue_bio)`，也就是电梯算法的入口。

## 2. blk\_queue\_bio

实际的处理函数是 `blk_queue_bio`。这个函数以设备的 `request_queue` 和要插入的 bio 作为参数，并且执行电梯算法。

- 执行 bounce 操作，就是在开启了 bounce 的情况下，将上层提交的 bio 拷贝一份再向下传递（可以支持重传），是否开启 bounce，取决于宏 `CONFIG_BOUNCE`。
- 检查完整性测试是否可以通过。是否开启该功能取决于宏 `CONFIG_BLK_DEV_INTEGRITY`。
- 如果队列允许合并，则调用 `blk_attempt_plug_merge`。这个函数不是针对全部 request 进行搜索合并，而是只针对要插入的 bio 搜索，看有没有可以合并的 request。如果有，则将该 bio 与该 request 合并。
- 如果队列不允许合并，则执行电梯算法进行总体合并，执行前要锁定 `request_queue`。

由于两种路径都要进行合并，一种是在添加的时候查找合并；另一种是电梯合并，在电梯合并时要对队列进行锁定。而老版本的 Linux 内核只有电梯合并一种路径。接下来将重点讨论电梯合并的情况，电梯合并的代码如下。

```

el_ret = elv_merge(q, &req, bio);
    if (el_ret == ELEVATOR_BACK_MERGE) {
        if (bio_attempt_back_merge(q, req, bio)) {
            elv_bio_merged(q, req, bio);
            if (!attempt_back_merge(q, req))
                elv_merged_request(q, req,
el_ret);
            goto out_unlock;

```



```

    }
    } else if (el_ret == ELEVATOR_FRONT_MERGE) {
        if (bio_attempt_front_merge(q, req, bio)) {
            elv_bio_merged(q, req, bio);
            if (!attempt_front_merge(q, req))
                elv_merged_request(q, req,
el_ret);
            goto out_unlock;
        }
    }
}

```

这几行最核心的合并代码到目前最新的 Linux 内核版本都没有发生变化。从字面意义上也很容易理解这段代码的含义，就是确定请求队列从前合并还是从后合并，确定了之后就尝试合并。前置合并和后置合并类似，区别是后置合并要改动 req->biotail，而前置合并只需要改动 bio，改动的方式又是一样的。

如果不可以合并（前后都不可以），则程序会继续向下执行，代码如下（代码为简化版）。

```

req = get_request(q, rw_flags, bio, GFP_NOIO); //获得一个空闲的 request
结构体
    init_request_from_bio(req, bio);           //用 bio 初始化这个结构体
    plug = current->plug;
    if (plug) {                                //如果现在队列处于 plug 状态，
简单地添加
        if (!request_count)
            trace_block_plug(q);
        else {
            if (request_count >= BLK_MAX_REQUEST_COUNT) {
                blk_flush_plug_list(plug,
false);
                trace_block_plug(q);
            }
        }
        list_add_tail(&req->queuelist, &plug->list);
        blk_account_io_start(req, true);
    } else {                                  //如果不是 plug 状态就立即执行
        spin_lock_irq(q->queue_lock);

```





```

        add_acct_request(q, req, where);    //把 request 添加
到队列 q 中
        __blk_run_queue(q);                //启动队列的执行
out_unlock:
        spin_unlock_irq(q->queue_lock);
    }

```

`add_acct_request` 这个函数会调用电梯算法的 `elevator_add_req_fn`，将 `request` 添加到电梯的队列中。

### 3. 其他重要的子函数

`elv_merge` 函数是电梯算法要执行的第一个函数，其首先尝试和 `queue->last_merge` 指定的上次合并的 `request` 进行合并计算。如果不成功就用哈希搜索 `request_queue` 进行合并尝试，仍旧搜索不到才调用电梯算法计算。注意，这一步仅仅是进行合并计算，也就是验证是否能够合并，具体的合并操作在 `blk_queue_bio` 函数中会根据 `elv_merge` 的返回值调用。传入的 3 个参数分别是 `request` 队列、作为返回值的标示可以合并的 `request` 和传入的 `bio`。也就是说，如果在队列 `q` 中找到了可以合并 `bio` 的 `request`，就将该 `request` 通过 `req` 传出。这个函数最后会调用电梯函数的 `elevator_merge_fn` 函数，看看电梯算法有没有合并的建议。电梯算法也只是通过计算判断能不能按照电梯算法的需求合并，并不是真正的进行合并。

`elv_bio_merged` 实际调用电梯算法的 `elevator_bio_merged_fn` 函数。具体的内容执行与具体的电梯算法相关。虽然之前有合并的数值计算，但是此处才是真正的合并方法。

`attempt_back_merge` 函数会首先调用电梯算法提供的 `elevator_latter_req_fn`。由于此时 `rq` 是之前 `bio` 要合并进入的请求，这个函数的作用就是找到队列 `q` 中的下一个请求，然后将这两个请求进行合并。怎么找到下个 `request` 是电梯算法的具体规定。但是合并之前可以做很多检查，例如现在是 `back_merge`，就需要检查下个 `request` 的物理地址是否刚好在 `rq` 之后。还需要检查两个 `req` 的方向是否一致，所作用的目标设备是否一致。两个 `bio` 是否是同一个（有可能发生重传，但是在这种情况下目的地址就可以过滤）。这里的合并参数调用了 `elv_merge_requests (elevator_merge_req_fn)`，也是电梯算法的函数。可以合并就合并两个请求的参数。例如 `sector` 数目、物理 `sector` 的数目等。然后执行真实的合并操作。最后再把已经执行完合并操作的请求放入队列。



`elv_merged_request` 函数实际调用的是电梯算法的 `elevator_merged_fn` 函数。在之前的操作理论上已经完成了合并。这些看起来重复的步骤其实是给电梯算法提供更多的选择。但是这一步进入的条件是上一步返回 0，也就是合并不成功。例如，如果 `elevator_latter_req_fn` 不返回有效的 request，这个函数就可以调用，而不用通用的合并框架代码。通用代码的最大缺点是只合并一个 `next`，如果想要一次合并多个 `next` 就可以在这里实现，但是这种情况确实很少，因为每个请求进来都会调用这个函数，除非新的 `bio` 可以导致两个本来不可合并的请求相邻，否则一次合并就够用。这里的进入条件并不是上次 `attempt_back_merge` 合并失败，而是 `attempt_back_merge` 发现需要合并，并且已经完成了自己的动作才会进入这里。也就是说，进入这里就意味着合并必须要进行了，这里只是在合并需要进行的条件下通知电梯算法，让其做出适当的内部调整。

如果发现不可与已有的 request 合并，将实际调用 `_elv_add_request` 函数。其插入位置有很多种：`ELEVATOR_INSERT_SORT`（默认）、`ELEVATOR_INSERT_FLUSH`、`ELEVATOR_INSERT_REQUEUE`、`ELEVATOR_INSERT_FRONT`、`ELEVATOR_INSERT_BACK`、`ELEVATOR_INSERT_SORT_MERGE`、`ELEVATOR_INSERT_SORT`、`ELEVATOR_INSERT_FLUSH`。我们只看第一种，这是大部分 `bio` 走的路径。这一种路径是首先将请求的 `hash` 合并到电梯算法的哈希表，以让电梯算法可以见到这个请求的存在，然后调用电梯算法的 `q->elevator->type->ops.elevator_add_req_fn(q, rq)`；进行实际的添加。

#### 4. plug 机制

如果当前的 `queue` 正在执行电梯算法，该 `queue` 就会处于 `plug` 状态。处于该状态的 `queue` 不会被真正的发送出去。这也是电梯算法的意义，电梯算法在执行时队列是要被锁定的，自然队列中的请求也不能交给下层处理。执行完电梯算法后会 `unplug`，队列流水线才可以正常执行。

#### 5. 总结电梯算法

由上文可以看到各个电梯函数在不同的时刻被调用，并且在被调用时很多电梯函数可以存在，也可以不存在。电梯函数结构体定义的代码如下。

```
struct elevator_ops{
    elevator_merge_fn *elevator_merge_fn;
```





```
elevator_merged_fn *elevator_merged_fn;
elevator_merge_req_fn *elevator_merge_req_fn;
elevator_allow_merge_fn *elevator_allow_merge_fn;
elevator_bio_merged_fn *elevator_bio_merged_fn;

elevator_dispatch_fn *elevator_dispatch_fn;
elevator_add_req_fn *elevator_add_req_fn;
elevator_activate_req_fn *elevator_activate_req_fn;
elevator_deactivate_req_fn *elevator_deactivate_req_fn;

elevator_completed_req_fn *elevator_completed_req_fn;

elevator_request_list_fn *elevator_former_req_fn;
elevator_request_list_fn *elevator_latter_req_fn;

elevator_init_icq_fn *elevator_init_icq_fn;
elevator_exit_icq_fn *elevator_exit_icq_fn;

elevator_set_req_fn *elevator_set_req_fn;
elevator_put_req_fn *elevator_put_req_fn;

elevator_may_queue_fn *elevator_may_queue_fn;

elevator_init_fn *elevator_init_fn;
elevator_exit_fn *elevator_exit_fn;
};
```

与判断是否可以合并相关的是 `elevator_merge_fn`、`elevator_merged_fn`、`elevator_merge_req_fn` 这 3 个函数。`elevator_merge_fn` 用于判断是否可以合并，是向前合并还是向后合并。`elevator_merged_fn` 是实际更新请求进行合并，如果实际进行了合并操作，就会继续调用 `elevator_merged_fn`。`elevator_merged_fn` 是在确定了向前合并还是向后合并后，调用的回调用来做本电梯算法内部数据的一些调整（根据是向前合并还是向后合并）。电梯算法的队列定义如下。

```
struct elevator_queue
{
    struct elevator_type *type;
    void *elevator_data;
```





```

struct kobject kobj;
struct mutex sysfs_lock;
unsigned int registered:1;
DECLARE_HASHTABLE(hash, ELV_HASH_BITS);
};

```

每个电梯算法都有队列，其中 `elevator_data` 存放电梯算法私有的数据，`elevator_type` 存放电梯算法提供的操作。由于每个 `queue` 对应一个电梯算法，每个电梯算法对应一个 `elevator_queue` 结构体，所以这个结构体的存在就是为电梯算法服务的。

最后定义了哈希表。这个哈希表是排序过的，以请求计算出 `key`，添加的部分是 `request->hash` 域。也就是说，每个新来加入队列的请求，其 `hash` 域都会被在这里添加。

当电梯算法被执行时，电梯算法只需要考虑这个结构体。当添加一个新的请求时，会将其首先添加到最后定义的 `hash` 中，然后会调用电梯算法的 `elevator_add_req_fn` 函数。如何组织这些请求取决于电梯算法的实现，但是都是组织在 `elevator_data` 中的。每个电梯算法都可以自由定义这个结构的用途。

也就是说，这里的请求会被添加两次。添加到 `hash` 的那次用于日后方便检索，而添加到 `elevator_data` 的那次用于服务电梯算法。在电梯算法运行处理时，其处理的对象就是 `elevator_data` 中由自己存放的数据。

## 6. noop 电梯算法

看一个最简单的 `noop` 电梯算法，这是 Linux 内核中可选的电梯算法，如图 11-5 所示。

```

root@ubuntu:/# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
root@ubuntu:/# █

```

图 11-5

可以看到当前 Linux 内核中支持的算法和当前磁盘所使用的算法。`noop` 电梯算法定义如下。

```

static struct elevator_type elevator_noop = {
    .ops = {
        .elevator_merge_req_fn      = noop_merged_requests,
        .elevator_dispatch_fn       = noop_dispatch,
        .elevator_add_req_fn       = noop_add_request,
    }
};

```



```

        .elevator_former_req_fn      = noop_former_request,
        .elevator_latter_req_fn     = noop_latter_request,
        .elevator_init_fn           = noop_init_queue,
        .elevator_exit_fn           = noop_exit_queue,
    },
    .elevator_name = "noop",
    .elevator_owner = THIS_MODULE,
};

static int __init noop_init(void)
{
    return elv_register(&elevator_noop);
}

static void __exit noop_exit(void)
{
    elv_unregister(&elevator_noop);
}

```

可以看出电梯算法接口的使用方法。一个结构体，然后启动时注册，关闭时解注册即可。电梯算法有很多操作，这个 `noop` 定义的只是一部分。但是也是精简到只留下必需的。

`noop_init_queue` 函数定义这种算法如何安排它的 `request queue` 队列。内容就是生成 `elevator_queue` 结构体并注册。`struct noop_data` 是 `noop` 算法挂载在电梯结构体上的私有数据，挂载在 `elevator_data` 上，这个数据仅仅是个 `list`。代码如下。

```

struct noop_data{
    struct list_head queue;
};

```

`noop_add_request` 非常简单，仅仅是将请求添加到电梯算法的队列 `elevator_data` (`noop_data`) 中。

前面讲到 `noop_latter_request` 和 `noop_former_request` 这两个函数，对应的电梯函数发生在 `bio` 合并进了请求之后，寻找下一个可以跟已经合并的请求进行合并的请求。前后类似。这步发生的条件是 `bio` 可合并且已合并到已有的请求。其返回的 `next` 是所请求的 `next`。

`noop_dispatch` 是把 `noop_data` 的第一个元素取出来，重新排序加入 `request_queue` 队列。排序的方法是 `sector` 的顺序。这是处理队列，在 `I/O` 调度算法执行结束后，



需要实际执行请求。调度算法执行的时候该请求不在电梯算法主程序的控制范围内。但是当调度算法执行结束时,该请求就通过本函数归还主程序的 `request_queue` 队列。`noop_merged_requests` 直接把 `next_request` 从上层的 `request_queue` 中删除。

## 7. deadline 电梯算法

deadline 电梯算法的定义如下。

```
static struct elevator_type iosched_deadline = {
    .ops = {
        .elevator_merge_fn = deadline_merge,
        .elevator_merged_fn = deadline_merged_request,
        .elevator_merge_req_fn = deadline_merged_requests,
        .elevator_dispatch_fn = deadline_dispatch_requests,
        .elevator_add_req_fn = deadline_add_request,
        .elevator_former_req_fn = elv_rb_former_request,
        .elevator_latter_req_fn = elv_rb_latter_request,
        .elevator_init_fn = deadline_init_queue,
        .elevator_exit_fn = deadline_exit_queue,
    },
    .elevator_attrs = deadline_attrs,
    .elevator_name = "deadline",
    .elevator_owner = THIS_MODULE,
};
```

这种算法也是比较简单的。算法的核心思想是请求的 `sector` 临近的合并,并且保证不临近的都有一个适当的延时,不至于饥饿,是标准的电梯算法。因为磁头移动的距离越短传输效率越高,但是总是这样移动就可能给远距离的请求带来饥饿。所以既要近距离移动磁头,又要保证远距离的请求不饥饿。要实现这个算法就需要两个结构体,一个是 `rb_tree`,另一个是 `FIFO`。`rb_tree` 用来查找 `sector` 最靠近的请求进行合并,而 `FIFO` 用来拿到该要超时处理的接近饥饿的请求。`rb_tree` 中的节点是用 `request->__sector` 组织的。

所以在添加操作时(`deadline_add_request`)会同时添加到 `rb_tree` 和 `FIFO` 这两个结构体中。在处理时要根据超时检查,也要兼顾处理 `rb_tree` 和 `FIFO` 这两个结构体。而在合并操作时,在大部分情况下是使用 `rb_tree` 的。





## 11.4 缓存层

### 11.4.1 BDI: 缓存设备

BDI 是对块设备层的内存支持, 相关代码页位于 `mm` 目录下。BDI 的全称是 `backing device info`, 后备设备是非易失性存储器, 但是这种存储器的访问速度都比较慢, 所以需要缓存。bdi 对应的结构体是 `backing_dev_info`, 这个模块完成的工作就是对于每个 BDI 设备都对其写入操作进行缓存, 然后在恰当的时间写入到 BDI 设备。需要注意的是, 每个磁盘都对应 BDI, 而磁盘上的文件系统可以使用 BDI, 也可以不使用 bdi。如果使用 BDI 就意味着本质上有了双层的 BDI。

由于这里使用的是恰当的时间, 这个时间点的选择就可以有多种情况。例如周期性的, 或者是内存使用到了一定程度时启动 BDI。这种方式启动的操作无疑是个内核线程 (也可以是工作队列), 但内核中采用了内核线程。在 2.6.30 Linux 内核版本以前, 这个叫作 `pdflush` 线程。而之后的 Linux 内核对性能进行了优化改进, 就变成了 `bdi-default`、`flush-x:y` 等多个线程。

这里考虑 Linux 内核新版本的结构。`bdi-default` 是 `flush-x:y` 的父线程。`bdi-default` 根据情况产生或销毁一个或多个 `flush` 线程。`flush-x:y` 中的 `x` 表示设备的种类, `y` 表示序号。这是 Linux 内核设备中 `major` 和 `minor` 的编号思路。当然, 对一个设备既有写也有读, 但是读的主要机制是预读。

BDI 的实现为链表。因为会有多个 BDI 设备 (磁盘等), Linux 系统习惯把多个同类设备组织成链表。由于 Linux 系统中链表的名字一般用结构体的第一个域 (链表域) 来表示。而 `backing_dev_info` 的第一个域为 `bdi_list`。代码中默认生成了两个全局的 `backing_dev_info` 结构体, 即 `default_backing_dev_info` 和 `noop_backing_dev_info`。而 4.02 版本 Linux 内核中只有一个 `noop_backing_dev_info`。

每个功能模块都有两种意义上的初始化, 一种是模块整体的初始化 (包括初始化全局变量); 另一种是模块所支持的实体的初始化。这个模块可能可以处理多个实体, 每个实体在被添加的时候也都需要初始化。`backing-dev` 就是这样的模块。

#### 1. `backing_dev_info`

`backing_dev_info` 结构体的代码如下。

```

struct backing_dev_info {
    struct list_head bdi_list; //bdi 设备列表
    unsigned long ra_pages; /*bdi 预读存储的页数*/
    unsigned int capabilities; /*设备能力*/
    congested_fn *congested_fn;
    void *congested_data; /*congested_fn 的数据*/
    char *name;
    unsigned int min_ratio;
    unsigned int max_ratio, max_prop_frac;
    atomic_long_t tot_write_bandwidth;
    struct bdi_writeback wb; /*回写数据结构*/
    struct list_head wb_list; /*回写数据结构的链表*/
    struct bdi_writeback_congested *wb_congested;
    wait_queue_head_t wb_waitq;
    struct device *dev;
    struct timer_list laptop_mode_wb_timer;
};

```

## 2. 初始化和注册

模块初始化，代码如下。

```

static int __init default_bdi_init(void){
    int err;
    bdi_wq = alloc_workqueue("writeback", WQ_MEM_RECLAIM |
WQ_FREEZABLE |
WQ_UNBOUND |
WQ_SYSFS, 0);
    if (!bdi_wq)
        return -ENOMEM;
    err = bdi_init(&noop_backing_dev_info);
    return err;
}

```

可以看出这个功能模块使用一个 writeback 的 workqueue，然后初始化一个 (Linux 内核老版本有两个) 全局的 bdi 结构体，这个初始化操作调用的是相关的实体初始化函数。而 4.02 版本 Linux 的内核代码只有 noop\_backing\_dev\_info 一个全局对象，所以只需要初始化一个。实体初始化函数有两个，即 int bdi\_init(struct

backing\_dev\_info \*bdi) 和 void bdi\_wb\_init(struct bdi\_writeback \*wb, struct backing\_dev\_info \*bdi)。

实体初始化函数主要设置 backing\_dev\_info 的参数, bdi\_wb\_init 初始化 wb 域对应的 bdi\_writeback 结构体。bdi\_init 会调用 bdi\_wb\_init 来完成它的初始化工作。

一般一个 bdi 设备就是一个分区。由文件系统调用注册函数 bdi\_setup\_and\_register 初始化和注册 bdi。而设备调用 bdi\_register\_dev 注册 bdi 设备。bdi\_register\_dev 是 bdi\_register 的简单封装。

## 11.4.2 页回收

- buffer.c:: block\_read\_full\_page, 这个函数使用一个 page 请求数据。
  - (1) 这个 page 如果有关联的 headbuffer 就使用, 没有就生成新的。
  - (2) 更新 buffer 和 page。
  - (3) 锁定 buffer。
  - (4) 调用 submit\_bh 启动读请求。
- buffer.c::submit\_bh 根据给定的 buffer head 生成 bio 结构体, 调用 submit\_bio 将其提交。
- blk-core::submit\_bio 做一个读写记录, 然后调用 generic\_make\_request 将请求提交给通用块层。

请求不止一个入口, 真正的页高速缓存位于 mm/filemap.c 中, 组织成了一个 radix 树, 可以根据 adree\_space 和 偏移来检索 (find\_get\_page) 或者添加 (add\_to\_page\_cache)、删除 (remove\_from\_page\_cache)、更新页 (read\_cache\_page)。

释放页是影响性能的关键, 也是会用到这部分功能的人所关心的问题。释放页首先调用的是 mm/filemap.c 中直接操作函数 try\_to\_release\_page。该函数释放本页上的所有缓存, 只是尝试释放, 但不一定能释放。

会触发回收的上级代码。

- 在 mm/vmscan.c 中有 shrink\_page\_list 函数, 会尝试回收 page。
- 在 mm/truncate.c 中有 invalidate\_complete\_page, 会回收 page。
- 在 mm/swap.c 中交换时会回收 page。
- 在 fs/splice.c 中 splice 函数会尝试“偷”走一个 page。



- 在 fs/buffer.c 中 block\_invalidatepage 会释放 page。

在 block\_dev.c 中有相关的定义，代码如下。

```
static const struct address_space_operations def_blk_aops = {
    .readpage      = blkdev_readpage,
    .writepage     = blkdev_writepage,
    .sync_page     = block_sync_page,
    .write_begin   = blkdev_write_begin,
    .write_end     = blkdev_write_end,
    .writepages    = generic_writepages,
    .releasepage   = blkdev_releasepage,
    .direct_IO     = blkdev_direct_IO,
};
```

理论上每个文件系统都会定义如何释放一个页的函数。所以这个函数在 ntfs 和 fat 中有可能不同。shrink 函数进入的次数比较频繁，而且进入一次后就会多次进入 releasepage 函数，原因很容易理解，shrink\_inactive\_list 函数是 shrink 的调用者，会遍历所有的 inactive 的页面来 shrink。再上层的调用函数是 shrink\_list，其既会回收活动的 page 也会回收不活动的 page。再向上是 shrink\_zone 和 shrink\_all\_zones 函数。

调用 shrink\_zone 的函数有以下 3 个。

- \_\_zone\_reclaim（该函数又由 zone\_reclaim 唯一调用）这个函数只在 page\_alloc.c 文件:: get\_page\_from\_freelist 中调用，作用是申请一个页，如果没有可用的页就触发收缩内存。
- balance\_pgdat 这个函数是由内核线程 kswapd 执行的，这个内核线程定期执行，回收一部分的 buffer 和 cache 的内存，默认是 5%。所以可以通过设置这个线程执行的时间或者提高这个线程回收内存的比例来使得内存回收得更快。
- shrink\_zones（此函数又由 do\_try\_to\_free\_pages 唯一调用，其又由 try\_to\_free\_pages 唯一调用（没有 cgroup），而该函数也是在 alloc\_Page.c 中需要内存时被触发，还有就是在 buffer.c 中触发 pdflush 之后，调用这个函数来回收一些内存），可见这可能不是我们看见的最多的那个。

拷贝大文件是一个受这里的算法影响非常大的应用场景，因为不可能所有的文件内容都存储在内存中缓存。可能一开始很多人会认为只需要开辟一块缓存，再进行文件复制就可以了，但是内核并不这样想。因为当我们实际调用 read 系统调用的

时候，内核的原则是要进行预读，提前将文件之后的一块内容读入。这是由于磁盘的一次读取的量多一些，相对于多次读取，每次少读更加高效。这个预读机制可以使用 `madvise` 给内核提建议。然而不但要预读，读完了之后内核还会倾向于保留读取到的数据在磁盘中，防止下次再次读取。也就是说拷贝文件这种流失操作对内核的内存管理机制几乎是一种攻击。内核在这方面有所思考，但是目前的实现仍然有不足之处，现在仍要依赖 `kswapd` 进行快速回收内存，或者手动通过 `proc` 文件系统进行回收，命令如下。

```
#回收文件内容缓存(Cached):
echo 1 > /proc/sys/vm/drop_caches
#回收文件目录和inode信息(Buffers):
echo 2 > /proc/sys/vm/drop_caches
#两者都回收:
echo 3 > /proc/sys/vm/drop_caches
```

### 11.4.3 缓存机制

内核中所有的页都会用来做文件缓存，当内存不够的时候再回收该部分缓存。因此页分为匿名页和文件缓存页。匿名页是进程使用的，文件缓存页用于文件的缓存。普通的内核读写都是要使用缓存层提供的缓存，但可以指定 `Direct I/O` 来绕过缓存机制，一般数据库系统都是直接 `I/O`，但直接 `I/O` 有对齐要求，操作麻烦，所以在个人编程时很少用到。

至于 Linux 提供的异步 `I/O`，则一定是使用的 `Direct I/O`，因为异步 `I/O` 要求立即返回，而经过缓存的 `I/O` 则是要阻塞等待文件更新到缓存后才会返回，所以不能被异步 `I/O` 采用。所以我们可以知道异步 `I/O` 完成的时候就是数据已经实际写入硬盘了（而同步 `I/O` 不一定）。异步 `I/O` 即使原则上不允许阻塞，但是由于其在实现时使用了锁，还是有可能阻塞的。

既然要在内存中缓存文件，有的文件很可能不能完整地缓存到内存中。所以就有了数据组织的问题。组织的方法是使用一个叫作 `buffer head` 的结构体作为一个文件的总体缓存情况的描述，该结构体是个 `list`。每个 `buffer head` 描述的文件缓存是一个缓存页的基树。基树的组织方式使得在查询缓存时更快。

现在就可以更加方便地理解内存的缓存状态信息了。命令如下。



```
root@ubuntu:~# cat /proc/meminfo
```

MemTotal: 2030492 kB //除去启动前被 BIOS 保留的内存, 剩下可供 kernel 支配的内存。这个值在系统运行期间一般是固定不变的

MemFree: 732560 kB //这个值一般比较小, 原因就是很多被用于文件缓存。这个值表示当前没有被使用的内存大小

MemAvailable: 1307848 kB //这个值并不等于 MemTotal - MemFree, 因为这个值还包含其他的在用到的时候可以被回收的内存。例如命名页的文件缓存 (Cached), slab 或者其他的缓存 (Buffers)。这个值是估算的, 并不是相加的和

Buffers: 253452 kB //一些网络传输或者虚拟设备之类的数据传输, 后面没有真实的文件, 这部分缓存就放在 buffer 里面。现在的内核还有 buffer 存储一些文件元数据, 比如目录 inode

Cached: 358048 kB //这部分就是内核中用于缓存文件内容的内存, 也就是缓存层的核心数据存储位置。包括的内容有很多, 但是都是要求基于文件的背后缓存, 例如 tmpfs (和基于 tmpfs 的 System V IPC); mmap 映射 (public) 或共享库、打开进程文件的 mapped 内存; 拷贝文件产生的没有映射的内存等

SwapCached: 21700 kB //这也是文件缓存的一种。但是那些既缓存到 swap 文件又还在内存中缓存的数据(一般是先缓存出去, 再加载回来)。如果 tmpfs 的内容被进行了 swap, 则该部分不再属于 Cached, 转而属于 SwapCached

```
Active: 493280 kB //Active(anon) +Active(file)
```

```
Inactive: 490744 kB //Inactive(anon) + Inactive(file)
```

Active(anon): 145424 kB //匿名页包括很多子类型。例如应用进程申请使用的内存就属于匿名页。Buffers 里面的也属于匿名页, Shared memory 和 tmpfs。简单地说匿名页就是没有对应具体文件缓存的页。这里的匿名页和 AnonPages 有所区别, AnonPages 没有记入 Shared memory 和 tmpfs, 这里的统计不包括被 mlock 的内存

```
Inactive(anon): 240548 kB
```

Active(file): 347856 kB //命名页就是有文件在磁盘, 也有内存中的缓存所占的内存。之前说过的文件预读会占用一部分, 交换分区也会占用, 还有其他的情况会占用

```
Inactive(file): 250196 kB
```

Unevictable: 0 kB //这里面包括了被 lock 的页, 但仅限于此, 例如还有 LRU\_UNEVICTABLE 的 lru 页面

```
Mlocked: 0 kB //被锁定的内存 mlock 调用
```

```
SwapTotal: 1046524 kB //swap 文件的总大小
```

```
SwapFree: 671500 kB //swap 的可用大小
```

Dirty: 144 kB //被标记为 dirty 的页面是要准备写回的。这个写回不但文件的写回, 还有把 swap cache 内存为 dirty 可以让交换分区进行交换

```
Writeback: 0 kB //正准备执行回写的内存。系统中全部的 dirty pages
```

```
= ( Dirty + NFS_Unstable + Writeback )
```



```

AnonPages:      370428 kB //匿名页。mmap 系统调用的 private 属于匿名页，而
public 映射属于 Cached。匿名页与进程相关，进程退出，匿名页释放

Mapped:         61264 kB //Cached 的页面有很多是没有被映射的，比如复制文件
的残留 cache 内存。但很多是被实际映射的，比如共享库、可执行文件、mmap 的文件。这个域就是
Cached 域中被 map 的内存空间大小

Shmem:          13448 kB //共享内存。tmpfs 文件系统作为非常常用的文件系统也
属于这个类别。tmpfs 实际挂载的大小可能会很大，但是很多内容会被交换入交换分区，实际的使用
量并不大

Slab:           222596 kB //slab 内存分配系统所有可分配的内存大小
SReclaimable:   165148 kB //可以被回收的 slab 内存
SUnreclaim:     57448 kB //不可以被回收的 slab 内存
KernelStack:    8432 kB //每一个用户线程都要对应内核栈
PageTables:     20484 kB //虚拟地址远比物理地址大。这个表用来将虚拟地址转
换为物理地址

NFS_Unstable:   0 kB //nfs 文件系统未写入磁盘的缓存页
Bounce:         0 kB //用于高低端内存的数据拷贝。现代 CPU 几乎都不使用

WritebackTmp:   0 kB
CommitLimit:    2061768 kB
Committed_AS:   2307980 kB
VmallocTotal:   34359738367 kB //一共被使用 vmalloc 接口申请的内存大小，可以
通过/proc/vmallocinfo 看到内存是被谁申请的

VmallocUsed:    0 kB //被实际使用的内存大小
VmallocChunk:    0 kB
HardwareCorrupted: 0 kB //系统检测发现的有问题的硬件内存区域大小（这些也会
计入 MemTotal 中）

AnonHugePages:  172032 kB //透明大页的单独统计
CmaTotal:        0 kB //CMA 连续内存分配器的内存分配情况
CmaFree:         0 kB
HugePages_Total: 0 //开启了大页之后，这 5 个域就是显示大页的内存使用情况的
HugePages_Free:  0
HugePages_Rsvd:  0
HugePages_Surp:  0
Hugepagesize:    2048 kB
DirectMap4k:     122752 kB //为了加速内存页的映射，很多大页被直接映射，而不使
用 TLB，这里就是显示直接映射的内存大小

DirectMap2M:     1974272 kB

```

值得注意的是，这里面你无法通过相加获得全部的内存。因为 `alloc_pages/_get_free_page` 作为内核内部重要的内存分配方式并没有被纳入统计，只能看到 `MemFree` 在减小。这个文件显示的内存状态相互之间的关系非常复杂，下面给出一个总体的关系。

- $\text{MemTotal} = \text{MemFree} + (\text{Slab} + \text{VmallocUsed} + \text{PageTables} + \text{KernelStack} + \text{Buffers} + \text{HardwareCorrupted} + \text{Bounce} + X) + (\text{Active} + \text{Inactive} + \text{Unevictable} + (\text{HugePages\_Total} * \text{Hugepagesize}))$ 。
- $\text{MemTotal} = \text{MemFree} + (\text{Slab} + \text{VmallocUsed} + \text{PageTables} + \text{KernelStack} + \text{Buffers} + \text{HardwareCorrupted} + \text{Bounce} + X) + (\text{Cached} + \text{AnonPages} + (\text{HugePages\_Total} * \text{Hugepagesize}))$ 。
- $\text{MemTotal} = \text{MemFree} + (\text{Slab} + \text{VmallocUsed} + \text{PageTables} + \text{KernelStack} + \text{Buffers} + \text{HardwareCorrupted} + \text{Bounce} + X) + (\sum \text{Pss} + (\text{Cached} - \text{mapped}) + (\text{HugePages\_Total} * \text{Hugepagesize}))$ 。

早期的一些操作系统没有 `MemAvailable`，你可以通过“`MemFree+Buffers+Cached`”得出一个近似值。

#### 11.4.4 缓存页的状态

由于缓存页也是页，其没有针对每个页的缓存用于专门定义一个结构体，而是与 `page` 结构体共享。所以在本层查询 `page` 结构体的状态就知道当前的页的缓存状态，与 `page cache` 相关的页状态主要有以下 4 种。

- `PG_uptodate`: 页缓存的数据是最新的。如果读此页，此页的数据将直接返回给读者，不需要从磁盘读。
- `PG_dirty`: 页缓存的数据是被写过，但是还没有写入磁盘的。当 `kswapd` 启动写入周期时，其会搜索到拥有该状态的页进行写入。
- `PG_private`: 表示的是页的从属是用来放 `buffer head` 的，并且该 `page` 结构体的 `private` 指针指向本页内存储的 `buffer head` 链表的头指针（一个页可以放很多 `buffer head`）。
- `PG_mappedtodisk`: 表示该页中缓存的所有数据都应在磁盘上有对应的块。这是由于有的写入会创建新的内存部分，此时该块内存在磁盘上没有对应的内

容，状态就是 `PG_dirty`，但是在 `PG_dirty` 状态时不一定是创建新的部分，还可能是修改（内容）。

所有的状态都分别占用一个内存位，因此各个状态同时存在是可能的。

## 11.5 文件系统

文件系统是用来存储文件的，而文件一定是有属性的。但是不同文件系统的属性可能不同，但也有共同的（例如创建时间、大小），而很多文件系统的属性（或者说是文件的属性）都可提供可选的功能，例如 `atime`（`access time`）、访问时更新，很多时候用户都不会用到这个属性所提供的功能，又由于其增加了 `iowait` 时间，所以很多优化都会将其关闭。而这个关闭是在文件系统层次上的。

也就是说，当使用一个文件系统时，首先要知道该文件系统都有何属性，哪些属性是可选的。然后根据自己的需求在挂载的时候打开或关闭属性。

不但是属性，还有文件系统本身提供的机制。例如日志功能、稀疏存储功能、完整性功能。这些功能有很多都是可以被打开或关闭的，对文件系统驱动的运行有影响，也都是在挂载的时候指定的。文件系统驱动会根据指定的功能开关决定是否在合适的时候执行操作。当然，这些特性也是不同的文件系统有所不同，但也有相同的。

这些属性和特性的打开都是在使用 `mount` 程序（或系统调用）时候要按照调用的格式指定的。

`dd` 命令可以用来从数据的意义上无格式地查看任何数据块。例如我们使用 `dd if=/dev/sda of=mbr bs=512 count=1` 就可以获得到 `sda` 磁盘的 MBR，然后就可以进一步分析这个 MBR 的格式。

`e2fsprogs` 内部有一系列的命令，可以用来查看和修改 `ext` 系列文件系统的一些高级内容。`mkfs` 系列命令可以用来制作各种文件系统，`fsck` 系列命令可以用来检查文件系统。

`sleuthkit` 系列工具是非常强大的文件系统检查工具集，例如 `fsstat` 可以列出文件系统的 `block` 和 `group` 情况；`lfs` 可以列出已经删除的文件；`icat` 可以查看 `inode`；`blkcat` 可以查看指定的 `block` 内容等。



## 11.5.1 文件系统的种类和选用

文件系统有很多种，Linux 内核是个“大杂烩”，它同时满足企业和个人的需求，文件系统也是如此，发行版也是如此。我们最常见的在 Linux 系统中的文件系统是 ext2、ext3、ext4，在 Windows 系统中的文件系统则是 ntfs、fat 和 exfat。这些文件系统既可以在企业用户中见到，也可以在个人用户中见到。

那为什么文件系统有的适合企业，有的适合个人呢？这就要从个人和企业的不同物理条件及需求上来说是了。对于企业来说，最重要的是数据安全，其次才是效率，然后是可扩展性，但是有的企业的应用也是非常注重效率的。而对于个人用户来说，最重要的是效率，因为个人的计算机处理能力一般有限。其次才是安全，然后才是可扩展性。而可扩展性的要求也远没有企业用户的需求大。在安全上，如加密、数据的一致性、数据的备份与恢复、服务热迁移等对企业来说是非常重要的需求，而对大部分个人用户来说可能一文不值。但有些功能虽然企业用户比个人用户需求大，但还是都可以接受的，例如日志系统，增加数据的一致性。很多用户担心设备突然断电后导致数据丢失会使用 UPS。但是如果你采用了 UPS，并且你的 CPU 处理能力有限，那么选用日志型的文件系统对你来说也不一定有意义。

在 Linux 系统下比较流行通用单机文件系统有 ext2、ext3、ext4，下一代可能变得通用的文件系统有 btrfs、NiLFS(2)和 exofs。分布式文件系统有 cifs (smb)、coda、afs。还有一些专用且比较常见的文件系统，如苹果系统的 hfs 和 hfsplus；Windows 系统的 ntfs 和 vfat，用于识别 iso 文件的 isofs。

## 11.5.2 拥有特殊功能的文件系统

在 Linux 系统下为了一些特定的功能目的，还实现了很多拥有特殊功能的文件系统，这些文件系统一般作为 Linux 系统功能的一部分而存在。

### 1. procfs

在前面所学的内容中我们大量地使用了 /proc 目录下的内容，这个目录是一个特殊的文件系统，启动的时候一般要挂在这个文件系统到 /proc 目录下。如果新的进程使用了新的 pid namespace，还会重新挂载一遍，因为旧的 proc 目录里包含了全量的

pid 目录。

proc 文件系统是 Linux 系统功能使用最频繁的功能性文件系统。里面主要是系统的各个维度的状态，包括每个进程的。此外其/proc/sys 目录下有很多对系统功能的控制能力。

## 2. sysfs

sysfs 也是几乎所有系统都会启动挂载的，默认在/sys 目录下主要是从设备的角度出发，包括设备的驱动、分类、状态等信息。相对底层的设备的信息查看和控制都在这里完成。

## 3. tmpfs

tmpfs 也是几乎所有系统都会挂在的，在/tmp 目录下。但是这个文件系统一般不会只挂载这一个。因为 Linux 内核的 System V IPC 也是依赖于 tmpfs 实现的，cgroup 的根目录也是使用 tmpfs 的。最近几年/run 目录开始出现，这个目录本身也是 tmpfs 文件系统，这个目录里的/run/lock（各个程序的 lock 文件）和/run/shm（共享内存）也是常用的 tmpfs 文件系统。tmpfs 使用内存作为文件系统，后面也可以有交换空间。

## 4. configfs

configfs 与 sysfs 类似，区别在于 sysfs 用于查看在内核中创建和销毁的组件，而 configfs 可用于用户空间创建和销毁内核对象。系统一般会默认挂载在/sys/kernel/config/目录下。命令如下。

```
mount -t configfs none /config
```

可以挂载这个文件系统，在这个文件系统中 rmdir 就是对对应的内核组件的创建和删除。而只要创建了一个目录（也就是创建了内核组件），里面对应的内核文件就会自动出现，像操作普通文件一样操作这些出现的文件就可以完成对对应内核组件属性的操作。不过对 configfs 文件的写操作要注意的是，要一次性全部写入。也就是你需要先把文件整体读取出来，然后修改，最后一次性写入。

当你 mount 了这个文件系统之后，用 ls 命令查看到了支持的子系统，就可以通过 mkdir 创建该子系统的条目了。在能看到支持的子系统之前，你必须首先加载这些子系统对应的模块。但是 configfs 明显很少被使用和支持，一般的设备都会在 sysfs 中直接提供配置修改的能力。所以很多时候 configfs 里面都是空的。

## 5. cgroup

cgroup 文件系统是 cgroup 功能的载体，其根目录是 tmpfs，cgroup 本身并不是严格意义上的文件系统，而是一种内核功能接口。使用 tmpfs 的文件形式为上层软件提供使用的接口。但是 mount 子系统的时候 type 会是 cgroup。

## 6. binfmt\_misc

在 Linux 系统下大部分二进制现在都是 ELF 格式的，但是使用这个文件系统就可以让 Linux 系统支持其他格式的二进制，当然包括 EXE 格式的二进制（但是系统调用都对不上）。命令如下。

```
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,relatime)

root@ubuntu:/# cat /proc/sys/fs/binfmt_misc/status //查看当前状态
disabled
root@ubuntu:/# echo 1 >/proc/sys/fs/binfmt_misc/status //启用模块
root@ubuntu:/# cat /proc/sys/fs/binfmt_misc/status
enabled
root@ubuntu:~#modprobe binfmt_misc //加载内核模块
root@ubuntu:~#echo ':Wine:M::MZ::/usr/bin/wine:' >
/proc/sys/fs/binfmt_misc/register //注册支持 EXE 格式执行的 wine 程序
root@ubuntu:~#./test.exe //直接执行 EXE 程序
root@ubuntu:/# echo -1 >/proc/sys/fs/binfmt_misc/status //关闭模块并且
清空已经注册的格式
```

也就是说，这个文件系统相当于调用 wine ./test.exe 来执行程序。不仅可以实现同样的功能，还能直接使用 bash 的首行和 ELF 的解析器段。例如在 bash 的首行添加#!/usr/bin/python，那么这个脚本的名字就可以直接在命令行中输入，bash 会负责找到/usr/bin/python 去执行脚本的内容。ELF 的 INTERP segment 里面的加载器也可以起到类似的作用。

## 7. autofs

以前的 Linux 系统使用/etc/fstab 文件来决定挂载什么文件系统。但是像 nfs、cifs 等网络文件系统，即便没有映射也会实际挂载，这就在一定程度上浪费了资源。然而在很多时候并不需要持续的挂载，这个时候 autofs 就可以做到。但一般情况下实



用性不大，在 docker 上有很多应用。

## 8. fusectl

用户端文件系统是一种可以在用户端实现文件系统驱动的文件系统，开源的 ntfs 就是使用 fuse 实现的，这个文件系统用于实际控制当前挂载的 fuse 文件系统，比如 fuse.gvfsd-fuse 是 gnome 的虚拟文件系统，它会单独挂载一个类型为 fuse.gvfsd-fuse 的文件系统。这个 fusectl 的实现还远远称不上完善。

## 9. kernfs

sysfs 里面的功能也越来越多，kernfs 文件系统的创建初衷是为了将 sysfs 文件中的一些功能提取出来并移到 kernfs 中。直到 3.14 版本的 Linux 内核才引入了这个功能集，因此现在极少应用。

## 10. debugfs

专门用于调试内核的虚拟文件系统，大部分子系统都没有使用这个，但是在内核空间提供了一种方法，让新的模块以这个方式向用户端输出内核的调试信息（当然，用户也可以自己在 proc 文件系统或者其他子系统添加条目），debugfs 的出现是为了统一调试输出。printk 不方便打印太多的数据，所以按需请求的 debugfs 有市场，Netlink 也不太适合这种场景。

debugfs 在有的实现中可以用来恢复文件，因为 ext2 的很多接口通过 debugfs 导出了，但是其对 ext4 的支持一般。当有高级需求的时候，人们还是会求助于专用的系统，而不是 debugfs。

## 11. securityfs

一般挂载在 /sys/kernel/security/ 目录下。大部分人不会用到这个文件系统，除非是系统的高级安全运维人员或者内核安全模块的开发者。这个虚拟文件系统直接在 vfs 上添加了一层封装，代码比较简单。它的意义在于它是专用于安全模块的配置和查看文件入口的。例如 apparmor 的内核模块会在这里创建目录，用户端的程序就可以在这里与内核模块配合通信了。

## 12. devtmpfs

这个文件系统是用来取代 devfs 的，devfs 的设计思路很好，但是缺少更新，后

来人们转向用户端的程序 udev 之类的，但是后来 devtmpfs 又出现了，完全取代了 devfs，现在是 dev 目录的标准文件系统。

### 13. devpts

在 Linux 系统下，大部分的用户登录都需要一个终端来进行输入输出的交互。正常情况下这个终端是 tty，硬件串口可以是 ttyS0，通过 usb 登录的是 ttyUSB0 之类的，这些都是设备驱动力的节点。但是当我们使用 SSH 登录远端的服务器时，也需要一个终端来输入和输出命令，这时内核给我们虚拟了一种终端，就是 pts。这个虚拟的终端是以文件系统的形式一般被 mount 在 /dev/pts 目录下的。这里面有不同的设备文件，每一个设备文件就代表一个用 SSH 登录的用户。你可以使用 who 命令查看到哪个用户在哪个 pts 上，然后使用 echo “hello” > /dev/pts/3 命令给这个用户的终端发送消息（假设这个用户在 3 号终端）。

### 14. hugetlbfs

大页文件系统就是在 Linux 系统下对大页机制的支持方式。一般使用 4KB 以上的大页才会需要用到。使用场景一般是高性能应用服务。

### 15. efivarfs

Linux 的启动系统现在开始使用 EFI (UEFI) 作为启动管理，这个文件系统是用来查看这个启动系统的参数的。但是使用场景非常少。

## 11.5.3 其他领域的文件系统

分布式文件系统的需求很早就存在。局域网中常见的 cifs (smb) 和 nfs 就是早期的满足这个需求的产品。在广域网范围内仍然有大量的分布式文件系统的需求，元老级的是 afs 和 plan 9。afs 是 1982 年为卡内基梅隆大学的大学档案管理设计的文件系统，plan 9 也是跟其差不多的时间由贝尔实验室的元老们开发的 9 号计划的一部分，设计思路非常激进。现在在 Linux 服务器上仍然有相当大的市场，应用的分布式文件系统有 gfs2、ocfs2 和 ceph。gfs 是谷歌开发的分布式文件系统，ocfs2 是 Oracle 开发并维护的文件系统，ceph 是红帽公司的官方维护的分布式文件系统。可见大公司的持续参与才是大型开源软件的活力的根本。

文件系统本身是不分嵌入式的和 PC 端的，只是不同的定位使得不同的文件系统被应用到不同的使用场景。嵌入式文件系统典型的需求就是“小”。例如在路由器中经常使用的 squashfs 和 cramfs，其压缩程度让人发指。还有专门为 flash 存储载体而设计的，如 jffs2、yaffs、f2fs、logfs、romfs 等。

还有加密专用文件系统 ecryptfs，它有些类似于 Windows 系统下的 bitlock，将整个文件系统加密。但是文件系统绝对不止这些，这是一些典型的例子。

### 11.5.4 文件系统的意义

那么为什么要有文件系统呢？文件系统存在的必要性是什么呢？SCSI 是一种磁盘命令集，USB 与 PCI 是一种传输数据的总线。在整个流程中还有一个没有涉及的问题，就是数据如何在磁盘上组织。我可以发一条读取的 SCSI 命令给磁盘，然而读取命令的参数是块，磁盘只能理解读取哪一块数据，我就将该块返回。换句话说，磁盘不知道文件、目录等结构的存在，只知道数据块的存在。那么在读取一个文件的时候，就需要有逻辑的将文件目录转换为对应的磁盘块的逻辑，这个转换过程就是文件系统最主要的工作。

文件系统在工作时完成目录文件到数据块请求的转换，也正是由于它的这种责任，数据如何在磁盘上存储也必须由其规定（否则其不知道如何转换）。组织数据的方式和转换请求的方式定义了不同的文件系统。

文件系统在组织上大同小异，主要分为分布式系统、单机文件系统和特殊文件系统。个人使用得最多的是单机文件系统。例如 ext4、ntfs 等。每个文件系统位于一个分区，但是一个磁盘上可以有多个分区。磁盘如何组织多个分区是有约定俗成的数据规范的（一个 512 字节的启动块，内部定义多个分区的位置），不属于任何文件系统的范围。

这些单机文件系统的统一特点是将磁盘分区划分为大小相等的块（一般是 1KB~4KB），然后将这些块组织成一个个含有相同块数的组。然后在每个组内包含一个个文件索引，每个索引指向了数据位于该组内的真实数据位置。另外，为了表征组内有哪些数据块还可以用，通常还会有一个位图。为了描述组的情况，一般有一个超级块。其实就是一个分层的树形资源组织方式。目录可以组织为普通文件，文件内容是下级文件的索引。也可以组织为特殊构造的实体。只要不遗漏信息，任



何组织架构都是可以的。现代的文件系统通常还包含了日志功能，在每次数据写入后要修改日志，没有修改日志的数据写入系统后会被忽视。这样就避免了因忽然掉电而导致的数据丢失。

### 11.5.5 文件系统的抽象：VFS

既然我们知道有如此多的文件系统，而内核的很多功能模块都需要同文件系统交互，所以不可能针对每一个文件系统都修改内核的其他部分。文件系统整体应该向外提供一个统一的数据结构和函数操作接口，这个接口叫作 VFS。

这个接口存在的依据是各个文件系统在上层抽象层次都需要是一样的。数据结构上都包含或者可以生成 3 个主要元素：超级块（整体描述整个文件系统分区的使用情况）、目录（里面是各个文件）、文件节点（存放文件信息）。文件数据的具体位置是由文件节点描述的。在操作上文件系统都是基本一致的，无非就是读写文件、重命名、添加、删除等通用的文件操作构成了所有文件系统的通用操作。

因此将 3 个数据结构和通用操作抽象出来，一起给内核其他模块提供使用时就构成了内核的 VFS 文件系统抽象层。各种文件系统驱动代码，本质上都是各自实现各自不同的这些操作的方式。

那既然都是操作同样的文件系统，用户端如何知道操作的是什么文件系统呢？在 Linux 的“哲学”里，所有设备资源的操作都在设备目录/dev 中，你所操作的文件一定是挂载在某个目录的某个设备上的。这个设备可能就是分区（也可以是回环）。于是你就对应地可以找到这个设备，查看这个设备的文件系统信息就知道了。也可以直接通过 VFS 读超级块的信息，从软件的角度查看磁盘信息。

正是因为内核对所有文件系统的这种统一的抽象，使得内核自己可以即使在没有实际文件系统时使用任意创建的模拟文件系统，只需要 3 种数据结构的组装即可。然而用途更大的是挂载思想。由于在内核看来每个文件系统都有一个根目录，所有该文件系统的子目录都可以通过根目录的 `dentry` 结构体向下递归查询来得到，而 `dentry` 又是位于内核的结构体，所以内核可以自由改变 `dentry` 的下级目录。比如改成某个文件系统的根目录。如此就是挂载。同样的思想，不但是根目录可以挂载，一个文件系统的子目录在理论上也是可以直接挂载的，然而内核并没有直接提供这种功能。通过各个文件系统的挂载，在一个根目录下，内核就可以将所有的存储系

统组装为一个目录树。目前各个 Linux 发行版都已经基本统一了这个根目录的结构，例如 dev、etc 等。

文件系统的作用不过是对存储方式的解析，这个解析过程相当于一个算法，输入文件的定位要求，返回定位的结果，然后由内核的 SCSI 命令部分将其转换为 SCSI 命令。所以其算法主体可以在用户空间执行，这就构成了用户空间文件系统 fuse。文件系统只是负责用它掌握的数据存储结构来计算具体的位置，而不执行实际的读写操作，真正的读写操作还是驱动程序在做。也就是说，文件系统本质上完成的是一个计算的工作，所以完全可以在用户空间计算。

## 11.5.6 ext4

### 1. 分块分组

文件系统的核心概念是分块，ext4 将块分为相等大小的 block，多个 block 组成 group。默认情况下一个 block 的大小是 4KB，一个 group 有 32768 个 block，也就是说一个 group 的默认大小是 128MB。但是这些参数是可以在制作文件系统的时候修改的。ext4 文件系统头部，如表 11-1 所示。

表 11-1

Group 0 Padding	ext4 Super Block	Group Descriptors	Reserved GDT Blocks	Data Block Bitmap	inode Bitmap	inode Table	Data Blocks
1024 bytes	1 block	many blocks	many blocks	1 block	1 block	many blocks	many more blocks

可以看到一个 group 内虽然有很多个域，但是每个域的大小也都是以 block 组织的。

表 11-1 是最全的 ext4 的格式，其中只有 Group 0 含有最开头的 Group Padding，这是用于存放操作系统的启动逻辑。ext4 Super Block 和 Group Descriptors 也不是每个 group 都有 Group 0 和一些其他被选中的 group 中会存在。例如要查看哪些 group 里放了 superblock 的备份，可以使用 `dumpe2fs /dev/sda2 | grep superblock` 命令来找到

哪些 group 中有 superblock 的备份，如图 11-6 所示。

如果 Group 0 的 superblock 坏了，也可以使用 `mount sb=32768 /dev/sda2 /mnt` 命令使用其他备份的 superblock 来挂载。

```
Primary superblock at 0, Group descriptors at 1-6
Backup superblock at 32768, Group descriptors at 32769-32774
Backup superblock at 98304, Group descriptors at 98305-98310
Backup superblock at 163840, Group descriptors at 163841-163846
Backup superblock at 229376, Group descriptors at 229377-229382
Backup superblock at 294912, Group descriptors at 294913-294918
Backup superblock at 819200, Group descriptors at 819201-819206
Backup superblock at 884736, Group descriptors at 884737-884742
Backup superblock at 1605632, Group descriptors at 1605633-1605638
Backup superblock at 2654208, Group descriptors at 2654209-2654214
```

图 11-6

ext4 还支持 `flex_bg` 模式，可以合并连续的多个 group 为一个，以支持超大文件和加速加载速度。ext4 还有一个特性是 `META_BG`，是出于 Group Descriptor 要放在一个 block 里的考虑。如果放不下，就需要将整个 ext4 的文件系统划分为若干个 `meta_bg`，每个 `meta_bg` 都有独立的 Group Descriptor。由于一个 block 可以放下的 Group Descriptor 数目足可以支持 256TB 的空间，所以这个功能一般不需要启用。还有一个组织数据的形式是 `extent`。`extent` 是 ext4 为了大文件内容尽可能地在磁盘中靠近所创建的概念。一个 `extent` 就是多个 block，也就是给一个新创建的文件尽可能大的空间，让后续的写入可以在连续的位置上。

## 2. inode

每个组里面都会有 inode table 和 inode bitmap。inode bitmap 用于表示 inode 的存在情况，而 inode table 则是真实的 inode 结构存放位置。每一个 inode 都代表一个文件，可以是文本文件、二进制文件或目录。inode 是不区别文件类型的，对于 inode 来说，它所代表的只是一些数据的存储块。

1~11 号的 inode 是在每个 group 中都存在的，都有特殊的意义，如表 11-2 所示。



表 11-2

inode	Purpose
1	有问题的 block 列表文件
2	根目录
3	User quota
4	Group quota
5	Boot loader
6	Undelete directory
7	Reserved group descriptors inode. ("resize inode")
8	Journal inode
9	The "exclude" inode, for snapshots(?)
10	Replica inode, used for some non-upstream feature
11	Traditional first non-reserved inode. Usually this is the lost+found directory. See s_first_ino in the superblock

并不是每个 inode 都实际对应后面的 block，有的文件小于 64 字节（还可以增大到 256 字节），文件内容就可以放在 inode 结构体内部，而不需要实际使用 block。

至于 block 如何使用，都是由 inode 结构体决定的。例如 xattrs 也是在 inode 结构体中的指针指向单独为这个文件分配的 block。这个文件系统为其他系统组件所增加的功能支持，例如 selinux，也都是通过 inode 结构体特定的域指向特定的 block 来完成的。

ext4 著名的日志系统是通过 inode 8 来支持的，也就是说 journal 本质上是分布在所有 group 中的一个 inode 的文件集合。

## 11.6 存储系统

### 11.6.1 存储形式

分布式存储系统与单机存储系统的界限不是特别明显，典型的分布式系统是很多地方拥有很多物理分离的存储设备的网络，并且容易扩展。典型的单机存储系统是在一个 PC 的磁盘上。

目前个人用户扩展磁盘的方法是使用光驱和 USB 接口的外置移动硬盘，最近市

场上出现了路由器上带 USB 接口，通过路由器上的 Samba 或 FTP 软件服务为本机扩展存储能力。前者叫 DAS，是直接硬件连接的。后者叫 NAS，是文件级的网络共享操作。还有一种为本机扩展存储的能力叫作 SAN。DAS 中的一个 read 调用会首先被文件系统驱动解析为一系列的 SCSI 命令，然后向下发送给 sata 或者 usb 连接的存储设备。NAS 中的一个 read 调用不经过本地的文件系统层，而是直接将文件请求通过网络转发到 NAS server（一般为 samba、nfs），NAS server 使用自己的文件系统驱动解析请求转换为 SCSI 指令发送给直接连接在 NAS server 的磁盘设备。而 SAN 则是一个 read 调用后，首先使用本机的文件系统驱动解析为一系列的 SCSI 命令，然后将这一系列的 SCSI 命令封包通过网络发送给 SAN server。SAN server 将这些 SCSI 命令解包并发送给与自己直连的磁盘设备。

也就是说 DAS 压根不在网络发送；SAN 是文件系统解析完后再发送到网络；NAS 是不经过文件系统解析就直接发送请求到网络。3 种不同的逻辑构成了 3 种分布式存储结构。这些都是为个人用户使用及拓展个人存储的方式。对应的还有不是服务于个人用户使用的，而是服务于外部任意用户使用的分布式存储系统（公有云和私有云），这种云存储一般是对用户呈现一个接口，仿佛只有一个硬盘，然而其在其后端可以动态地连接很多个跨地域的存储设备，带有负载均衡和数据保护的一整套系统。

DAS 常用的物理链路是 SCSI、SAS、SATA、USB。SAN 一般用于存储网络，使用的网络是光纤 FC 或者以太网、AoE（其他种类慢慢在被淘汰），传输 SCSI 命令的机制叫作 iSCSI。NAS 则是在现有的局域网上（一般是以太网）通过 cifs、nfs 等网络文件系统提供。

SAN 一般指专用网络，例如以太网、FC、AoE 等，是独立于其他网络单独存在的。但也可以使用以太网，你如果要与局域网复用也是可以的，只要你能忍受带宽的挤占。SAN 网络的最新宠是 iSCSI，其将 SCSI 命令在 IP 网络上发送，大部分使用以太网介质。由于 Linux 系统的 sg 模块可以发送原生的 SCSI 命令给磁盘，所以这个机制在 Linux 系统上是被支持的。Windows 7 系统以后都内置原生的 iSCSI Initiator，在 Linux 系统中可以安装 scsi-target-utils 作为 iSCSI target，即可以非常简单地像使用 Windows 系统访问本地磁盘一样访问 Linux 的磁盘。

## 11.6.2 存储格式

前面讲到的都是数据存储的物理形式，没有涉及太多数据存储的格式。分布式文件系统是一种兼具形式和格式的存储方式，但是分布式存储系统不一定需要使用分布式文件系统，分布式文件系统也可能使用在 SAN 网络中。存储数据的格式是根据存储数据的内容来选择的。有非结构化、半结构化、结构化这 3 种。我们可以将平时使用到的文件存储方式都称为非结构化的，虽然它们也有结构；xml 是典型的半结构化存储；数据库是结构化存储。

数据库有很多种，但都是组织为有特定格式的一系列数据的集合。它存在的根本目的是数据的存储、查询和修改。数据库本身可以在一台计算机（或多台计算机）组成单独的服务器，提供数据服务。然而这种服务不属于存储的扩展，而是数据的扩展。这种服务像 Web 服务一样，是一种互联网服务，只是服务的内容是数据，不同于 DAS、NAS、SAN 和云存储这些存储能力的扩展。虽然在很多情况下两者能达到相同的目的。DAS、NAS、SAN 和云存储的存储介质上一般都有文件系统，对用户提供的服务是存储本身，而数据库系统所存储的存储介质上不一定有文件系统，对用户提供的服务是抽象的数据服务，不涉及硬件。

## 11.6.3 分布式存储系统

从存储的意义上讲，分布式存储系统根据要存储的数据内容的不同分为分布式文件系统、分布式表格系统、分布式键值系统、分布式数据库。其中与 DAS、SAN、NAS 等同于同一个层次的是分布式文件系统，它描述了数据的组织形式。分布式键值系统、分布式数据库、分布式表格系统都是在已有的数据组织形式的基础上设计上层存储和访问方式。是一整套的数据接口和封装。

集群是一组相互独立的，通过高速网络互联的计算机组成的集合。集群一般可以分为科学集群、负载均衡集群、高可用性集群这 3 大类。科学集群是并行计算的基础。它对外就好像一个超级计算机，这种计算机内部由十至上万个独立处理器组成，并且在公共消息传递层上进行通信以运行并发应用程序；高可用性集群是指当集群中的一个系统发生故障时，集群软件迅速做出反应，将该系统的任务分配至集群中其他正在工作的系统上执行，通过消除单一故障点和节点故障转移功能来提供



高可用性，次节点通常是主节点的镜像；负载均衡集群是指将服务请求分摊处理到集群中的多个节点上。如软件型 LVS、硬件型 F5。在实际生产环境中，这 3 种集群相互交融，如高可用性集群也可以在节点之间均衡用户负载。

### 1. 分布式文件系统

由于分布式文件系统具备解析文件系统的功能，又需要组织成一个整体，所以其具备 SAN 的特性（文件系统逻辑在本机，SCSI 命令在网络）。所以一般分布式文件系统都运行在 SAN 网络上。

在用途上分布式文件系统是用于作为典型的文件系统存储非结构化的数据（文件），它还有一个很重要的用途就是作为分布式表格系统的持久层。分布式表格系统像表格存储的关系型数据库，但是又不同于分布式数据库系统。分布式数据库是一个数据库的拓展，拓展能力有限，除非是专门为分布式设计的数据库。可见的数据库系统大部分是单机存在的，或者是存在多个做负载均衡。而分布式表格系统是一个分布式的整体，其分布式本身就是系统的一部分。

分布式文件系统主要要考虑和解决的问题有以下 8 个。

- 如何组织各台电脑（一般为 C/S）？
- 如何避免系统访问“瓶颈”？
- 如何做负载均衡？
- 如何组织数据（例如一个文件拆分，同时写入多个节点）？
- 如何处理节点的不可用性和可用性（上线和掉线的问题）？
- 如何回收存储空间（移动数据使数据紧凑，或者垃圾回收）？
- 如何保证数据的完整性（保存多个备份）？
- 如何处理并发访问？

所有的分布式文件系统无非是针对这几个问题提出了不同的解决方法。

### 2. gfs

gfs 是谷歌的文件系统，主要为谷歌的内部使用优化。在谷歌的内部部署了大量的 gfs 文件系统节点，很多作为底层的存储结构服务于 Google Bigtable 分布式表格系统。被广泛使用的 Hadoop 系统的底层就是使用的 hdfs 文件系统，该文件系统是在用户空间用 Java 写的，但是实现的逻辑却是 gfs 的。也就是说，hdfs 是 gfs 的通用版本的实现，gfs 是基础。最新的版本是 gfs2，运行在 san 上，后面都说“gfs2”。

还有值得思考的一点是，gfs 是一个分布式文件系统，而 hdfs 则是用来组件云存储系统的云文件系统。虽然本质上差别不大，但是其在可扩展性和跨地域处理等问题的处理上必然有区别。还有一点就是 gfs2 是红帽公司开发的，并不是谷歌开发的，gfs 被开源后，任何人都可以修改，红帽公司在很多人对 gfs 做了很多改进后，集大成地推出了 gfs2，并且提供了一整套用于搭建 gfs2 可用存储集群的工具，这个套件叫作 RHCS (REDHAT CLUSTER SUITE)。

gfs 最显著的特点就是可以单机存在。与所有其他文件系统一样，无非就是一系列 inode 和一系列存储块。一个文件（目录）是一个 inode，在 gfs2 中，日志也是以文件的形式存在的 (inode)，gfs 则是专用的存储空间。

本地的大部分文件系统的写操作因为考虑突然断电的问题而都开始支持日志功能了，更何况是不知道另一个节点什么时候会掉线的网络存储架构。所以 gfs 对日志的支持是必要的。gfs 中是 3 层的架构，master 管理文件到 chunk server 的映射 (chunk 是 64MB 大小的数据库，没有格式，但是组合多个 chunk 为一个连续的文件就具备了文件的格式)，chunk server 负责管理 chunk、对 chunk 进行备份和查询修改。也就是说，用户向 master 询问到哪里可以找到对应文件的数据，master 指定 chunk server 为用户提供响应，之后用户则可以直接从 chunk server 那里获得需要的信息。chunk server 具备管理能力必须是在 master 的授权之后，这个授权有一定的时间限制，到期了就要续租。

### 3. 解决分布式系统通用问题的方法

对于 gfs 分布式文件系统需要解决的问题的回答如下。

- 组织：采用 C/S 架构，一个 chunk master 负责维护总体的文件存储信息（目录和 chunk 基本信息）、chunk 副本的位置信息、文件到 chunk 之间的映射。多个 chunk server 负责持久化地记录 chunk 副本的位置信息，处理用户的数据请求。更多的存储节点被 chunk server 调度，用于存放实际的数据。可以看出是 3 层的存储架构。其中第三层存储一般是直接连接在 chunk server 上的磁盘。
- 瓶颈避免：用户向 master 查询存储信息，然后再向查到的存储数据对应的 chunk server 直接请求数据。并且会缓存这种对应关系，之后的访问不需要查询 master。chunk server 会决定使用主数据或者是哪个副本进行响应 (chunk server 会维护副本的一致性)。

- 负载均衡。
- 组织数据：一份数据完整地存储于一个节点，但可以有多个节点存储该数据的拷贝。
- 可用性：每一份数据在多个节点并行保存多份拷贝（默认为 3 份）。
- 数据回收。

(1) 只采用追加操作，极大地精简逻辑，但是会带来很多废旧数据。

(2) 掉电机器再次上电，发现其存储的内容与其他数据拷贝不一致等很多情况，都会触发整个系统的数据回收机制。数据回收的方法无非就是移动有用数据和清理无用数据。

(3) 当删除数据时，并不是直接将数据删掉，而是只是先标记数据，待系统不忙的时候再慢慢启动回收机制进行回收。

- 数据完整性。

(1) 保存多个备份容易，维护多个备份困难。例如在写的时候 gfs 客户端会将要写的数据发送到所有拷贝节点后才触发写操作。多个备份必须要同时写成功。

(2) 还有就是容错能力，这就需要日志系统的支持，这也是支持原子操作的手段。

(3) 还有 master 的备份，叫作 shadowmaster。

(4) 提供快照功能使数据容易回溯。这里的快照设计得非常精巧，直接利用写时复制的特性，增加对 chunk 的引用即可。这样后续在对 chunk 的修改时都会发现原始的 chunk 有人使用，而新建的 chunk 在新的 chunk 上修改。快照时刻的 chunk 得以永久保存，随时回溯。

一个文件系统必然服务于创始人的某些目的，谷歌的文件系统的设计也就必然服从于谷歌的需求。例如其追加写、延迟删除、64MB 的 chunk 大小。虽然磁盘利用率很低，但精简了逻辑，提高了效率。虽然有回收机制，但是这是以空间换时间的做法，不适合小型的分布式系统。

可以自己确定有多少份拷贝，然而分布式系统中节点的断线和上线的频率不一样，有的甚至是区域不一样。这种“一刀切”的方法无法适应所有的情况。但通过调整这个值可以适用于大部分情况。虽然多份拷贝会影响速度，但 gfs 在设计上已经尽量减小了多份拷贝对速度的影响了，适用异步的更新拷贝。chunk server 会负责更新拷贝，而用户对数据的操作却不会因此而阻塞。但这都是在预定的机制下进行优化的方法。



gfs 是为通常不稳定的情况设计的。例如在选择初始存储节点时，会选用负载较低的节点，而副本又会避免在同一个机架（避免同时崩溃），还有可能在使用过程中重新复制副本（例如损坏）。例如在所有机器都在一个房间放置的分布式系统中直接使用这些策略这件事是值得商榷的。

这些也都不能算是缺点，只是如果通用就无法最高效的特用，最高效的特用就无法通用，这是所有系统的难题。在使用中要时刻告诉自己：现在的性能不是最好的。

#### 4. RHCS (REDHAT CLUSTER SUITE)

通过上面的介绍我们可以发现，使用 gf 不能像使用传统分区一样，必须要有专门的知识作为支撑点。谷歌也有这个号召力让用户专门为使用它的系统而更新自己的客户端。其实这也就决定了其不是为个人用户使用而开发的文件系统。gfs 文件系统的存在必然伴随着存储服务或者云存储、云数据服务。

因此就需要配套的以 gfs 为底层依托提供云存储服务的套件程序。红帽公司优化了对 gfs 的修改，并且创造了配套的程序后推出了 RHCS 服务套件。

RHCS 提供了集群系统中 3 种集群构架，分别是高可用性集群、负载均衡集群、存储集群，更加侧重定位于高可用集群。但是不要理解为其包含这 3 个组件，而只是这个系统实现了这 3 个概念。其实原始的 gfs 也有可用性检测、负载均衡的思想，但是做得不如专用的工具优秀。

高可用集群是 RHCS 的核心功能。当一台主机出现异常时，应用可以通过 RHCS 提供的高可用性管理组件自动且快速地从 一个节点切换到另一个节点，节点故障转移功能对客户端来说是透明的，从而保证应用持续地对外提供服务，这就是 RHCS 高可用集群实现的功能。然而这种方式也完全可以用前端的负载均衡和后端热备方案来替代，所以实际使用得不多，在 nginx 大行其道的今天，负载均衡才是核心话题。

RHCS 通过 LVS (Linux Virtual Server) 来提供负载均衡集群，LVS 是一个基于 IP 的负载均衡技术，LVS 由负载调度器和服务访问节点组成，通过 LVS 的负载调度功能，可以将客户端请求平均分配到各个后端服务节点，同时还可以定义多种负载分配策略，当一个请求进来时，集群系统根据调度算法来判断应该将请求分配到哪个服务节点，然后由分配到的节点响应客户端请求。同时 LVS 还提供了服务节点故障转移功能。也就是当某个服务节点不能提供服务时，LVS 会自动屏蔽这个故障节

点，接着将失败节点从集群中剔除，同时将新到此节点请求平滑地转移到其他正常的节点上。而当此故障节点恢复正常后，LVS 又会自动将此节点加入到集群中。而这一系列的切换动作，对用户来说都是透明的，通过故障转移功能保证了服务持续且稳定的运行。

RHCS 通过 gfs 文件系统提供存储集群功能，gfs 允许多个服务同时读写一个单一的共享文件系统。存储集群通过将共享数据放到一个共享文件系统中，消除了应用程序间同步数据的麻烦。gfs 通过锁管理机制协调和管理多个服务节点对同一个文件系统的读写操作。

CCS (Cluster Configuration System) 主要用于集群配置文件的管理和配置文件在节点之间的同步。CCS 程序运行在集群的每个节点上，监控每个集群节点上的单一配置文件/etc /cluster/cluster.conf 的状态。当这个文件发生变化时，CCS 会将这些变化更新至集群中的每个节点上，时刻保持每个节点的配置文件同步。Cluster.conf 是一个 XML 文件，其中包含集群名称、集群节点信息、集群资源和服务信息、fence 设备等。

CMAN (Cluster Manager) 是一个分布式集群的管理工具，运行在集群的各个节点上，为 RHCS 提供集群管理服务。它用于管理集群成员、消息和通知。它通过监控每个节点的运行状态来了解节点成员之间的关系。当集群中某个节点出现故障时，节点成员的关系将发生改变，CMAN 及时将这种改变通知底层，进而底层做出相应的调整。CMAN 根据每个节点的运行状态计算出一个法定节点数作为集群是否存活的依据。当整个集群中有多于一半的节点处于激活状态时，表示达到了法定节点数，此集群可以正常运行。当集群中有一半或少于一半的节点处于激活状态时，系统就认为集群没有达到法定的节点数，此时整个集群系统将变得不可用。CMAN 依赖于 CCS，并且 CMAN 通过 CCS 读取 cluster.conf 文件。

DLM (Distributed LockManager) 是一个分布式锁管理器，为集群提供了一个公用的锁运行机制。DLM 运行在每个节点上，gfs 文件系统通过锁管理器的机制同步访问文件系统的元数据。CLVM (Clustered Logical Volume Manager) 通过锁管理器同步更新数据到 LVM 卷和卷组。DLM 采用对等的锁管理方式，突破了单个节点失败而需要整体恢复的性能瓶颈。

通过栅设备可以从集群共享存储中断开一个节点，切断 I/O 以保证数据的完整性。当 CMAN 确定一个节点失效后，它在集群结构中通告这个失效的节点，fenced



进程将失败的节点隔离，以保证失败的节点不破坏共享数据。它可以避免因出现不可预知的情况而造成的“脑裂”（split-brain）现象。“脑裂”现象是指当两个节点之间的心跳中断时，两台主机都无法获取对方的信息，此时两台主机都认为自己是主节点，于是对集群资源（共享存储、公共 IP 地址）进行争用、抢夺。Fence 的工作原理是当由于意外原因导致主机异常或宕机时，备用机会首先调用 Fence 设备，然后通过 Fence 设备将异常的主机重启或从网络上隔离，释放异常主机占据的资源。当隔离操作成功后，返回信息给备用机，备用机在接到信息后，开始接管主机的服务和资源。RHCS 的 Fence 设备分为两种，即内部 Fence 设备和外部 Fence 设备。内部 Fence 设备有 IBM RSAII 卡、HP 的 ILO 卡、IPMI 设备等；外部 Fence 设备有 UPS、SAN switch、Networkswitch 等。

当节点 A 上的栅过程发现 C 节点失效时，它通过栅代理通知光纤通道交换机将 C 节点隔离，从而释放占用的共享存储。当 A 上的栅过程发现 C 节点失效时，它通过栅代理直接对服务器做电源的打开和关闭的操作，而不是执行操作系统的开关机指令。

rgmanager 主要用来监督、启动、停止集群的应用、服务和资源。当一个节点的服务失败时，高可用集群服务管理进程可以将服务从这个失败节点转移至健康节点上，这种服务转移能力是自动透明的。RHCS 通过 rgmanager 来管理集群服务，rgmanager 运行在每个集群节点上，在服务器上对应的进程为 clurgmgrd。

在 RHCS 集群中，高可用性服务包括集群服务和集群资源两个方面。集群服务其实就是应用，如 APACHE、MySQL 等；集群资源有 IP 地址、脚本、ext4 文件系统、gfs 文件系统等。

在 RHCS 集群中，高可用性服务是和一个失败转移域结合在一起的。由几个节点负责一个特定的服务的集合叫失败转移域，在失败转移域中可以设置节点的优先级。主节点失效，服务会迁移至次节点，如果没有设置各个节点的优先级，集群高可用服务将在任意节点间转移。

RHCS 根本不关心底层使用的文件系统存储结构，其关心的只是每一台机器本身的管理与调度。所以 RHCS 系统下的文件系统完全可以全部采用 ext4 文件系统，当然也可以采用分布式的 gfs 文件系统。

那么 gfs 文件系统存在的意义是什么呢？在分布式存储系统中都可以使用单机文件系统完成。首先我们要知道 gfs 文件系统本身就可以作为单机文件系统使用。





而 gfs 文件系统本身和 RHCS 的一些特性重合，例如负载均衡。然而 gfs 文件系统提供了单机存储集群提供不了的特性，比如集群快照、高效的追加操作、默认的备份能力。当然，你可以用 RDBD、RAID 等其他机制提供备份能力，但是这些特性是 gfs 文件系统自带的，并且从架构上决定了其成本会非常低。用外置工具组装一些 ext4 文件系统为集群就如同成立了一个多方联合作战的联盟军。而使用 gfs 文件系统作用集群文件系统如同成立了一个单一参与方的集团军，两者之间的战斗力还是有差别的。而且使用 ext4 文件系统或其他文件系统组成的集群，更像一种 NAS 的扩展，而不是一个云存储系统。



## 12

## 第 12 章

## 虚拟化与云

软件行业正在经历一场变革，一开始虚拟化和云是独立发展的，但是随着发展的深入，云越来越需要虚拟化提供的稳定的工作环境和可伸缩的特性。虚拟化也越来越需要云来拓展其规模。在这个融合的过程中，Linux 也引进了很多特性以适应行业的变化。

在虚拟化的发展过程中，被个人广泛使用的是运行在宿主操作系统上的虚拟化操作系统，例如 VMware、VirtualBox。在企业的应用中，早期也使用这种方案，但是由于太过重量级，对服务器硬件的性能损耗巨大，所以 VMware vSphere 这种提供底层虚拟化方案的也有一定的市场。但是在 Docker 出现后，一切都改变了，虚拟化全面向容器演变。容器既轻量级，对性能的损失又小，还提供了隔离。微软也提供了 app-v 这种轻量级的容器。

## 12.1 常见的虚拟化方案

### 1. 硬件模拟器

最复杂、最彻底的虚拟化技术是硬件模拟器。硬件模拟器在主机系统上创建硬件 VM，完全模拟硬件的功能。硬件模拟器的主要缺点是速度太慢，因为我们可以 Intel 的 CPU 上模拟 ARM 的 CPU，所以执行的指令不是真实地把指令交给硬件，而是所有的硬件指令都由软件模拟完成。优点是可以在 Intel 的机器上跑 power 等非



Intel 的架构程序。所以硬件模拟器的主要用途是用来开发固件，也就是在硬件还没有出来的时候，先模拟这个硬件，固件开发者就可以提前进行固件开发了。或者是用于学习和研究。比较常用的两个模拟器是 Bochs 和 QEMU。

## 2. 全虚拟化

全虚拟化相当于一个最底层的操作系统，这个操作系统的唯一功能就是调度不同的操作系统来分时占用硬件，所以其有极高的性能。例如 VMware 的核心产品 vSphere 就是一个全虚拟化方案（或者说是半虚拟化），还有 VMware Workstation 和 VirtualBox。

由于全虚拟化对于客户操作系统应该是无感知的，所以宿主系统的调度系统就需要付出更多的代价确保客户操作系统的无感知，缺少客户操作系统的配合，很多事情就会比较难做。所以在使用 VMware 的时候，一般需要额外安装一个 VM Tools 的工具包。操作系统虚拟化理论上性能是最低的，但是在实际应用中却是应用最广泛的。因为大部分人都是首先有宿主主机，然后需要其他系统的时候就在宿主主机上安装虚拟机。而企业也是在购买的服务器上已经预装了 Linux 系统，很多都有服务器专用的内核模块，大部分虚拟机也就直接在已有的 Linux 系统上安装了。很多开发者通常的工作模式是宿主系统是 Windows，虚拟机是 Linux 或者其他版本的 Windows。

## 3. 半虚拟化

半虚拟化就是针对全虚拟化的性能损耗过大的问题，让客户操作系统辅助支持一些操作。运行在半虚拟化系统上的操作系统是知道自己是虚拟机的，也知道如何与底层的系统进行交互来辅助底层调度系统完成任务。例如是否需要更多时间片、是否需要传输数据等需要协商的操作就相对容易，所以半虚拟化执行的速度与裸机相当。

半虚拟化在 Linux 中比较常用的系统是 UML (The User-mode Linux Kernel)、Xen 和 KVM。Xen 是非常知名的虚拟化方案，但也正是因为实际直接使用硬件，所以物理上半虚拟化有多少资源就只能卖多少资源，不能超售。Xen 虽然性能卓越，但是要求一个特制的底层内核。KVM 由于集成到内核主线，因此只需要使用通用的内核，不需要额外的定制。目前在高性能系统级虚拟化领域，KVM 几乎已经取代 Xen。





#### 4. 操作系统级的虚拟化（容器）

操作系统虚拟化就是我们在服务器上常见的虚拟化方案。比如 Docker、OpenVZ、Linux VServer 是常见的传统操作系统虚拟化方案。例如 Linux VServer 是直接使用同一个内核，但是通过修改内核代码来模拟出多个用户。被内核主线普遍应用的是轻量级的 LXC，LXC 就是在用户层级的虚拟化，该机制仅利用内核提供的 cgroup 和 namespace，为用户创造出一个仿佛独立的系统空间。然而实质上用户只是看到独立，其实还是公用的，也不需要做代码的转换，只是将资源在内部做了划分。LXC 在刚出现的时候也没有得到重视，直到基于 LXC 的 Docker 技术的出现，直接引爆了 LXC 背后的技术，决定了未来虚拟化技术的走向。

Docker 之所以能成功，主要是因为 LXC 无论在实际工程上还是在效率上都绝对高效，但是 LXC 所面对的问题是是否有可移植性。LXC 的配置在不同的机器上不能方便地在统一的环境进行发布。而 Docker 最重要的改进就是提供了统一的发布环境和规范，使得进程在统一的环境下发布，而又利用并增强了资源隔离。

#### 5. 库级虚拟化

例如 wine、LxRun 这种通过某个库就可以模拟一个系统的 API 的方式来提供一个系统的兼容性服务。操作系统不过是一系列函数的集合，无论是 Windows 还是 Linux，它们的区别就在于接口不同实现的逻辑不同，归根是代码不同。既然只是代码不同，那么在 Windows 上有而在 Linux 上缺少的代码，我们完全可以自己补上。只是完整地实现一样的函数是非常不明智的，大部分情况下只需要让它们的函数接口一样，函数执行的功能一样即可。这种模拟通常是调用本地系统已经提供的类似功能，例如 Windows 下的 open 函数和 Linux 下 open 函数中间需要的只是一个封装。

#### 6. 虚拟化的当下与未来

在非常大型的对外提供完全独立虚拟机的服务领域，Xen/KVM 目前是绝对的主流。例如阿里的 ECS 也是使用 Xen/KVM，亚马逊的 EC2 目前还在使用 Xen。使用 Xen 是从性能的角度出发，而使用 KVM 是从便利性的角度出发。

在企业的内部，当一个企业有几万台服务器时，企业一般会选择 Docker 这种容器化的解决方案，例如京东的很多服务器就已经全面推行了 Docker，各大互联网企业也都是正在或者已经使用了 Docker。其轻量、统一、隔离、工具丰富，Docker 目前在 server 程序发布上几乎无可替代。



所以对于大部分人来说, Docker 无论是从学习的角度还是从使用上来说, 都几乎是不二选择。

## 12.2 虚拟文件系统

虚拟文件系统并没有专用的特指, 而是某些文件系统的特性特别适合用于虚拟化, 或者说这些文件系统是专门为虚拟化而设计的, 典型的是 AUFS 和 overlayfs。但是虚拟化, 例如 Docker 还可以使用传统的文件系统的机制, 例如 device mapper 和 btrfs 都可以被 Docker 作为底层的存储系统, 然而这两种机制更多的是作为通用文件系统而存在的。Device mapper 所提供的磁盘管理能力正在逐渐被 btrfs 所取代。

### 1. AUFS

AUFS 是 Docker 出现时候的首选文件系统。AUFS 是一个融合作用的文件系统, 可以把多个文件夹的内容融合到一个文件夹内, 并且可以指定不同文件夹的权限。在他之前还有 unionfs, unionfs 是一个使用了用户端文件系统驱动接口 fuse 的文件系统, 所以效率比较低。而 AUFS 则是在内核中实现, 所以效率相对 unionfs 较高。

AUFS 一个重要的特性是其在合并各个目录的时候, 如果指定了某个目录只读合并, 那么在合并之后的目录中对这个文件进行修改是仍然可行的, 但是原来的文件并不会改变。对于只读挂载的目录, AUFS 采用了写时复制的策略, 将原来的文件拷贝一份作为仅存在于挂载之后文件系统上的文件。所以这里的只读是指原文件的只读, 并不是挂载之后的目录的只读。

Docker 就是利用 AUFS 的这种特性进行文件系统的文件隔离。但是由于 AUFS 刚刚出现的时候有三万多行的代码, 被要求严苛的 Linux 本人拒绝其进入内核, 即使作者努力修改, 最终也没有被允许, 但是一般都是发行版自己打补丁或者使用者亲自打补丁。而随着 overlayfs 的出现, AUFS 在性能上逐渐暴露出不足, 虽然很多 Docker 系统仍然使用 AUFS, 但是 overlayfs 已经成为趋势, 并且 overlayfs 被 Linux 接纳入内核主线程 (3.18-rc2), 改名为 overlay, 所以未来 overlay 取代 AUFS 几乎是板上钉钉的事情。



## 2. overlayfs

传统的方式是同一台电脑上多个进程使用同一个操作系统，看到的是共同的目录结构。当进程变为云进程时，由于其不是被自己放到电脑上执行的，所以这个进程的可信度就有很大问题。可以使用传统目录结构的方法为每个进程限定一个根目录，但各个进程运行所需要的库文件却无法为每个进程都做一个拷贝。

正式高度弹性的云服务提出了一个需求，即能否共用一套基本目录，但是每个用户对目录的修改不影响其他人。答案就是 overlayfs。这个文件系统在挂载时需要制定上半部分和下半部分目录。而这个目录有已有的任何文件系统的目录。最终呈现的 overlayFS 目录是两个目录的合并。其中制定的下层目录不会被真实修改，所有对下层的修改都是拷贝到上层隐藏下层。是一种写时拷贝思路的文件系统级实现。

这样可以将库文件等通用文件目录作为下层，而工作目录作为上层提供给不可信的进程。进程则没有办法对关键文件造成任何实质的破坏。本质上 overlayFS 并不是一种文件系统，而是提供了 vfs 操作接口的一种转换方式。类似于数据库中视图概念。正是这种概念的创造满足了云的需求。

因为 overlayfs 的接口转换的特性，下层的文件系统可以是任何的文件系统，并且不需要可写的权限。overlayfs 有一个很重要的特性就是当上层的文件（不包括目录）与下层的文件重名时用上层的文件名，而上层没有定义重名文件时用下层的文件名。所以这个特性几乎是为多个上层虚拟机共享下层的不变的二进制基础环境而设计的，并且允许上层对下层提供的二进制进行替换。

Linux 的 overlayfs 还支持多个底层文件系统共用一个上层文件系统。这样可以创造多层次的层叠目录结构。使用如下命令可以完成一个 overlayfs 的 mount。

```
mount -t overlay overlay  
-olowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

Mount 之后，merged 目录内就含有合并之后的实时的目录结构。当你在 upperdir 目录中更改、创建或删除一个文件时就会体现在 merged 目录下的完整结构中。Work 目录是 overlayfs 内部使用的，是必须提供的，并且要求和 upperdir 在同一个文件系统中的空目录中。另外，后来的 overlayfs (3.19) 做了增强，允许叠加多层，而不是之前的只有两层，这主要是为了满足 Docker 的多层栈的需求，命令如下。

```
mount -t overlay overlay -olowerdir=/lower1:/lower2:/lower3 /merged
```





如果这样做就不需要像两层那样指定 `workdir` 和 `upperdir` 了。因为这个 `lowdir` 从右到左向上叠加, `lower1` 已经是最上层, `lower3` 是最下层。

## 12.3 cgroup

`cgroup` 是现代 Android 的基础, 最初也是 Android 内部的人员提出的, 后来在 Linux 内核内实现。通过 `cgroup` 可以将定额的系统资源 (如 CPU、内存等) 分配给特定的一组进程。默认情况下, 编译内核时打开 `cgroup` 的系统中所有进程位于同一个 `cgroup`, 就是根, 这个 `cgroup` 享有所有的系统资源。

可以通过 `cgroup` 文件系统建立一个新的 `cgroup`, 然后配置这个新的 `cgroup`, 配置的内容包括为其分配进程、分配资源等。这个创建和分配的所有过程都是 `cgroup` 文件系统通过 `shell echo` 写进文件完成的。可以使用 `mount -t cgroup` 命令查看已经挂载的 `cgroup`, 如图 12-1 所示。

```
root@ubuntu:/cgroups# mount -t cgroup
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/lib/systemd/systemd-cgroups-agent,none=systemd)
cgroup on /sys/fs/cgroup/net_cls,net_prio type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls,net_prio)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/pids type cgroup (rw,nosuid,nodev,noexec,relatime,pids)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpu,cpuacct)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
root@ubuntu:/cgroups#
```

图 12-1

之前 Linux 发行版的 `cgroup` 并没有被广泛 (默认) 部署, 而需要手动创建, 现在的系统一般都有默认的 `cgroup` 结构。所以我们先分析已有的 `cgroup` 目录结构, 然后再从零组建。

如今 `cgroup` 的应用领域不断扩大, 各个发行版纷纷开始默认打开 `cgroup` 进行进程的调度和资源规划, 并且用户端的 LXC 虚拟化方案也是使用 `cgroup` 分配资源实现的用户端模拟虚拟运行环境。也正是这个最初不被看好的 LXC (比起 Xen、KVM), 由于 Docker 技术的出现, 而开始被广泛使用在云计算中。使用 `mount -t tmpfs` 命令查看已经挂载的 `tmpfs`, 如图 12-2 所示。

```
root@ubuntu:/sys/fs/cgroup# mount -t tmpfs
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=203052k,mode=755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
tmpfs on /run/lock type tmpfs (rw,nosuid,nodev,noexec,relatime,size=5120k)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,mode=755)
tmpfs on /run/user/116 type tmpfs (rw,nosuid,nodev,relatime,size=203052k,mode=700,uid=116,gid=116)
tmpfs on /run/user/1000 type tmpfs (rw,nosuid,nodev,relatime,size=203052k,mode=700,uid=1000,gid=1000)
```

图 12-2



我们可以看到, cgroup 的根目录是以 tmpfs 的形式进行挂载的, 而 `mount -t cgroup` 中显示的很多目录, 例如 `systemd`、`pids` 等目录则是不同的 cgroup 子系统。

cgroup 本身是分层级的, 一个根层下面像一棵树一样可以分很多层。每一层的 cgroup 文件系统目录下都有该层对应的资源配置文件。这些可以配置的资源都是 cgroup 子系统。这里所介绍的内容并不是指一份使用手册, 而是一个用于辅助理解 cgroup 是什么、它是怎样工作的说明。

## 1. CPU 子系统

CPU 子系统用于控制 cgroup 中所有进程可以使用的 CPU 时间片。附加了 CPU 子系统的 hierarchy 下面建立的 cgroup 的目录下都有一个 `cpu.shares` 文件, 对其写入整数值可以控制该 cgroup 获得的时间片。例如在两个 cgroup 中都将 `cpu.shares` 设定为 1 的任务, 将有相同的 CPU 时间, 但在 cgroup 中将 `cpu.shares` 设定为 2 的任务, 可使用的 CPU 时间是在 cgroup 中将 `cpu.shares` 设定为 1 的任务的可使用的 CPU 时间的两倍 (同一层级)。cgroup 的 CPU 子系统, 如图 12-3 所示。

```
root@ubuntu:/sys/fs/cgroup/cpu# ls
cgroup.clone_children  cgroup.sane_behavior  cpuacct.usage  cpu.cfs_period_us  cpu.shares  release_agent  tasks
cgroup.procs          cpuacct.stat          cpuacct.usage_percpu  cpu.cfs_quota_us  cpu.stat     notify_on_release
```

图 12-3

这里要引入层级的概念。如图 12-4 所示的 CPU 子系统 (cgroup 树型结构), 这个子系统内部拥有两个层级。第一个层级是最外层的, 第二个层级是里面的目录: `init.scope`、`system.slice`、`user.slice`。在 CPU 子系统的跟目录下只要创建一个目录就会自动生成这些标准文件, 创建目录的动作也就是创建了下个层级的一个分支, 还可以在有些目录的下面继续创建目录, 也就形成了第三层级。也就是说 cgroup 层级是一个树型结构。

```
root@ubuntu:/sys/fs/cgroup/cpu# ls init.scope/
cgroup.clone_children  cpuacct.stat  cpuacct.usage_percpu  cpu.cfs_quota_us  cpu.stat  tasks
cgroup.procs          cpuacct.usage  cpu.cfs_period_us    cpu.shares        notify_on_release

root@ubuntu:/sys/fs/cgroup/cpu# ls user.slice/
cgroup.clone_children  cpuacct.stat  cpuacct.usage_percpu  cpu.cfs_quota_us  cpu.stat  tasks
cgroup.procs          cpuacct.usage  cpu.cfs_period_us    cpu.shares        notify_on_release
```

图 12-4

CPU 子系统是通过 Linux CFS 调度器实现的, 可以说 Linux CFS 调度器的实现虽然是绝对公平的, 但实际上却是为不公平地使用 CPU 而准备的。CPU 子系统调度 CPU 的访问控制, 这里有两种调度模式: CFS (Completely Fair Scheduler) 和 RTS

(Real-Time scheduler), cgroup 下同时支持两种调度模式, 但是 RTS 的方式只在进程中采用了 RTS 调度算法的时候才有效。

例如, 如果想让一个 cgroup 在 1 秒内能有 0.5 秒的 CPU 使用时间, 设置 `cpu.cfs_period_us` 为 1000000 (1s), 设置 `cpu.cfs_quota_us` 为 500000 (0.5s)。`cpu.shares` 是一个大于等于 2 的整数值, 指定了相对系统上所有 CPU, 本 cgroup 使用 CPU 的权重。`cpu.shares` 为 2048 的 group 可使用的 CPU 资源是 1024 的 2 倍。`cpu.stat` 可以统计 CPU 的使用状态。

而对于 RTS 调度的进程 `cpu.rt_period_us` 和 `cpu.rt_runtime_us` 的设置就可以生效。详细的各个文件的使用方法可以见 man 文档。

`cpuacct` 子系统自动生成 cgroup 中任务所使用的 CPU 报告。`cpuacct.usage` 中是该 group 及其子 group 的 cpu 总使用时间 (纳秒); `cpuacct.stat` 中是该 group 及其子 group 的 CPU 的用户和内核态的分别使用时间; `cpuacct.usage_percpu` 中是该 group 及其子 group 的 CPU 分别使用时间 (纳秒)。

`cpuset` 子系统为 cgroup 中的任务分配独立的 CPU (在多核系统中) 和内存节点。`cpuset.cpus`: 绑定该 group 的 CPU 节点, 如绑定该进程可以使用 4、5、6、17、18 这 5 个 CPU, 写入文件的数据格式就是 “4-6,17,18”。

## 2. memory 子系统

memory 子系统可以设定 cgroup 中任务使用的内存限制, 并自动生成由哪些任务使用的内存资源报告。memory 子系统是通过 Linux 的 resource counter 机制实现的, 代码如下。cgroup 的 memory 子系统, 如图 12-5 所示。

```
struct res_counter {
    unsigned long long usage; //记录资源的当前使用量
    unsigned long long max_usage; //使用过的最大资源量
    unsigned long long limit; //资源最大限制, 分配超过此限制的资源将会导
致失败

    unsigned long long soft_limit; //资源软限制, 是可以超过的
    unsigned long long failcnt; //资源分配失败的次数
    spinlock_t lock; //自旋锁
    struct res_counter *parent; //形成资源统计组
};
```



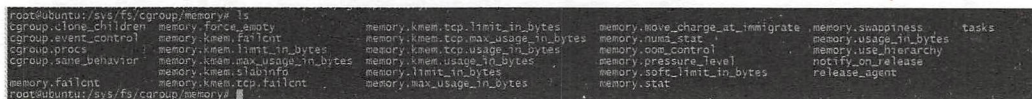


图 12-5

memory 也是一个非常复杂的系统，从名字上可以看出大部分文件的含义，修改对应文件就能达到要求的结果。运维人员可能会比较关心 memory.failcnt（达到内存限制（memory.limit\_in\_bytes）的次数）和 memory.memsw.failcnt（达到内存+swap 限制（memory.memsw.limit\_in\_bytes）的次数），还有 memory.stat（统计内存使用状态）。

与其他的内核系统一样，使用一个组件的时候通常都要是显性的，所以很容易想到 res\_counter 的使用需要 struct res\_counter 被放在对应的数据结构中作为组件，然后在每次申请资源的时候调用对应的调用，代码如下。

```
int alloc_abc(...)
{
    if (res_counter_charge(res_counter_ptr, amount) < 0)
        return -ENOMEM;
}

void release_abc(...)
{
    res_counter_uncharge(res_counter_ptr, amount);
}
```

这样就可以在资源申请与释放的同时进行 res\_counter 的设置与统计。

### 3. blkio 子系统

这个子系统为块设备设定输入/输出限制，比如物理设备（磁盘、固态硬盘、USB 等）。该子系统提供了两种控制 I/O 方式。

#### (1) 基于权重

每个 group 都可以设置一个数值，根据数值的不同，系统分配相应的 I/O。

- blkio.weight: 一个 100~1000 的数值。全局的权重；
- blkio.weight\_device: 一个 100~1000 的数值，指定设备的 I/O 权重。这里指定的权重值会覆盖全局的权重值。

以上两个值中同一个 group 只能存在一个。

## (2) 基于速度

每个 group 都有一个最大的速度，该 group 的进程 I/O 不能大于这个速度。

`blkio.throttle.read_bps_device`: 指定该设备上的最大读速度 (bytes/s);

`blkio.throttle.read_iops_device`: 指定该设备上的最大读 I/O (I/O read/s);

`blkio.throttle.write_bps_device`: 指定该设备上的最大写速度 (bytes/s);

`blkio.throttle.write_iops_device`: 指定该设备上的最大写 I/O (I/O read/s);

`blkio.throttle.io_serviced`: 记录设备 I/O 操作总数;

`blkio.throttle.io_service_bytes`: 记录设备读取总数;

其他配置项。

`blkio.reset_stats`: 对当前文件写入一个整数可重置当前所有统计数据;

`blkio.time`: 指定设备的 cgroup 控制的 I/O 访问时间 (ms);

`blkio.sectors`: 指定设备的扇区操作数;

`blkio.io_service_time`: 制定设备的 I/O 工作时间 (ns);

`blkio.io_wait_time`: cgroup 等待 I/O 的时间;

`blkio.io_merged`: 被合并的 I/O 请求;

`blkio.io_queued`: 被 cgroup 放到队列的 I/O 请求;

## 4. 其他子系统

`devices` 子系统是通过提供 `device whitelist` 来实现的, `devices` 子系统通过在内核对设备访问的时候加入额外的检查来实现。而 `devices` 子系统本身只需要管理好可以访问的设备列表就可以了。

`freezer` 子系统可以挂起或者恢复 cgroup 中的任务。`freezer.state` 可能读出的值有 3 种, 其中两种是 `FROZEN` 和 `THAWED`, 分别代表进程已挂起和已恢复(正常运行)。还有一种可能的值为 `FREEZING`, 显示该值表示该 cgroup 中有些进程现在不能被 frozen。当这些不能被 frozen 的进程从该 cgroup 中消失时, `FREEZING` 会变成 `FROZEN`, 或者手动将 `FROZEN` 或 `THAWED` 写入一次。

`memory` 子系统设定 cgroup 中任务使用的内存限制, 并自动生成由哪些任务使用的内存资源报告。`net_cls` 子系统使用等级识别符 (classid) 标记网络数据包, 可允许 Linux 流量控制程序 (tc) 识别从具体 cgroup 中生成的数据包。`net_prio` 子系统提供了一种动态控制每个网卡流量优先级的功能。

还有 `ns` 名称空间子系统。`ns` 子系统是一个比较特殊的子系统。`ns` 子系统没有

自己的控制文件，而且没有属于自己的状态信息。ns 子系统实际上是提供了一种同命名空间的进程聚类的机制。具有相同命名空间的进程会在相同 cgroup 分组中。

同一个子系统在同一个 cgroup 中只能出现一次。所有这些 cgroup 子系统都是以系统资源的纵向角度组织的，但是其又提供了横向组织的能力。在一个 cgroup 下挂载任何一个子系统都是针对当前的 cgroup 进行控制。可以说 cgroup 的设计是用最小的接口提供了最大的灵活性，但是这也使得 cgroup 的管理比较困难。cgroup v2 相对 cgroup v1 进行了巨大的重构，但是直到 4.5 版本的内核才完整地支持了 cgroup v2。所以在选用 cgroup 的时候要注意 cgroup 的有效性问题，如果已经使用了很新版本的内核，就可以直接使用 cgroup v2。

## 5. 在用户端使用 cgroup

如果在你的内核中编译时打开了 cgroup，在用户端要使用 cgroup 为应用程序分组分配资源，你还可以安装 cgroup 的用户端控制程序：libcgroup，这个程序是一个服务（centos），你可以做出 cgroup 规划，系统启动的时候会根据规划配置脚本为你建好 cgroup 架构，还提供了几个方便地操作 cgroup 的脚本。Ubuntu 下 cgroup 有个专门的工具集：cgroup-tools，但是 cgroup 几乎所有的操作也都可以轻松地直接使用文件系统完成。当然也可以什么都不安装，因为 cgroup 可以直接原生操作 cgroup 文件系统完成使用。通过 `mount -t cgroup -o debug cgroup /cgroup` 等命令进行操作，此时就会在 /cgroup 下生成很多默认的文件，如果在这个目录下建立文件，就意味着你建立了子 cgroup，用 `ls` 命令查询一下你建立的文件就会发现，里面的内容与根目录下自动生成的大致相同。cgroup 文件系统通过这种方式完成树形的层次组织。cgroup 挂载命令如下。

```
mount -t cgroup
mkdir cgroup
mount -t tmpfs cgroup_root ./cgroup
mkdir cgroup/cpuset
mount -t cgroup -ocpuset cpuset ./cgroup/cpuset/
mkdir cgroup/cpu
mount -t cgroup -ocpu cpu ./cgroup/cpu/
mkdir cgroup/memory
mount -t cgroup -omemory memory ./cgroup/memory/
```

建立一个 cgroup 并不能真正完成工作，你需要为其分配可用的资源并将不同的



进程放进去。当建立第一个 cgroup 时，系统会把所有的进程都放进去。也就是你会在 /cgroup/tasks 的 cat 文件中发现所有的进程（所有相关的配置和使用方法并不是固定的，Linux 在变化）。如果你在下级的 cgroup，例如 /cgroup/test 的 task 文件中将一个进程的 PID echo 到文件，就会发现这个进程并没有从上层 cgroup 的 task 列表中消失，而是同时出现在了下层 test cgroup 的 task 文件中，这就表示该进程现在归 test cgroup 控制。但由于 test cgroup 是下层的 cgroup，所以进程依然存在于上层的 cgroup，而同一层的 cgroup 之间却不能含有相同的进程。

一个属于某个 cgroup 的进程的子进程都会自动属于同一个 cgroup。可以看出 cgroup 是分层次的，这个层次叫作层级。而每个 cgroup 内部都可以针对 CPU、网络、磁盘等资源进行配置，这些可以配置的内容叫作子系统。

我们知道实时进程对 CPU 的使用很可能是独占的，内核为我们提供了不从调度算法本身，而是通过使用 cgroup 限制实时进程使用 CPU 的方法：RT Scheduling 和 RT Throttling。为何不在调度本身设计呢？因为实时的概念只是尽快抢占 CPU，而不涉及什么时候释放（直到更实时的）。实时的策略是对的，但是如果出现 bug，将会有很大的概率导致系统卡死。在 cgroup 文件系统中，使用 `cpu.rt_period_us` 和 `cpu.rt_runtime_us` 可以限制本组的实时进程的 CPU 占用。

还可以使用 cgroup 将用户端的进程分成一个个的组，然后以组为单位设置调度方法。例如使用 Fair Group Scheduling（该机制利用内核的 CFS 调度机制），实现各个分组之间的公平调度算法。

还可以使用 cgroup 的 `cpuset` 功能为每个进程组规定使用哪些 CPU，以及每个 CPU 的亲和度。还可以针对特定的程序组对它们的内存进行限制、统计和强制释放。还可以针对 I/O 操作，设置分组的 I/O 优先级。将后台执行磁盘操作的程序放到较低优先级的分组中，前台需要高响应速度的放在较高优先级分组中。这样既可以有效地利用 I/O 吞吐量，还可以不影响前台软件的使用性能（`echo 100 > /cgroup/low/blkio.weight`）。

## 12.4 Docker

Docker 发展至今也已经变化了很多，相信未来也会继续变化。本书使用的 Docker

版本是 1.12.1。这里简单介绍 Docker 所依赖的框架性的原理，而没有深入介绍诸如 swarm、docker link 等 Docker 特有的特性。

## 1. 存储

Docker 的下层存储需要做到文件系统的修改隔离，并且文件系统中应该有各自独立的私密文件。所以会采用的存储引擎都要求是分层的。Docker 的部署架构分为 container 和 image。image 类似于我们常见的操作系统，但它不是操作系统。Docker 最大的成功就在于其提供了可以供部署应用程序的统一的环境，这个 image 就是统一的环境。而一个 image 环境可以支撑多个 Docker container 在其上运行，所以就要求 image 本身是只读的，但是每个 container 由于要进行修改，所以必须是可写的。这个架构要求完全符合之前所说的 overlayfs 等分层文件系统所提供的功能。

也正是由于这种设计，所以在安装了一次 image 后，后续只需要启动新的 container 就可以了，同一台机器上再有 Docker 的实例需求就并不需要再次下载 image 了。

这种复用被 Docker 更进一步的利用。主要是将 image 分为多层。我们自己使用一个 image，完全不会意识到这样做的意义，但是当有很多个 image 需要管理的时候，例如 Ubuntu server 16 和 Ubuntu destop 16 可能共享很多二进制，但是只是配置不同。它们在 image 层次上就可以做到很多复用，这种复用还包括很多配置上的。事实上，如果你下载一个 Ubuntu 16.04，你会发现你下载了 4 个 layer，然而整个系统的大部分大小几乎都集中在其中一个 layer 上（不是所有的 image 都如此）。因为 Docker 的 image 并不是只为静态的组织进行设计的。更多考虑到了实际的使用，用户实际修改了 image。几乎大部分公司在拿到 Docker 提供的 image 之后都会添加自己的配置，临时文件或者是二进制作为自己私有的 image。因此，大小大部分集中的那一层就是底层，而之上的几层 image 就是常用文件的一些覆盖。例如 /var、/tmp、/etc、/run、/sbin，这样即使之后 Docker 更新了 image，也几乎只是更新最底层的 image，而用户私有地修改则在更高的层次，从而只需要更新底层。

在 image 之上的 container 则是 rw 的，它的定位是程序实际执行的环境，某个 container 私有的修改可以放在这里（例如一个产品的后端程序的日志输出），但是公司整体的私有 image 则是在 image 的层级中组织的，而每个服务在发布服务程序本身的时候通常也是进一步私有地修改 image，将自己的二进制放入 image 后再发布到目标机器上直接运行。

除了分层的存储架构，Docker 还提供了直接访问的存储方式，叫作 volume。

volume 可以绕过分层文件系统，直接写入物理磁盘，并且它提供了多个 container 共享数据的功能。

## 2. 网络

Docker 上使用的网络协议栈代码上必定是外层内核的协议栈代码，但是网络设备的存在形式可以不同，而且数据也是通过 network namespace 隔离的。可以使用 NAT 的方式把所有 container 都作为内网，使用内网 IP。或者是用 host 的方式直接赋给外部 IP，直连内核协议栈。Host 的方式是最高效的，但是与其他的 container 没有隔离，bridge (nat) 的方式提供了比较常用的组织，NAT 的方式可以隐藏内部服务和划分主机内部 container 之间的网络结构。

除此之外，使用 dpdk+vSwitch 作为网络分发的方式比较高级，需要额外的编程，也是可以采用的方案。

这样的安排提供的绝对不是网络虚拟化的作用，因为各个 container 还是共享物理设备和路由表的。实际的 Docker 还使用了 linux network namespace 用于隔离路由表，使用 VETH 用于创建虚拟物理设备。也就是说，通常一个 container 的网卡都是对应内核的一个虚拟的物理设备，而各个物理设备之间由于使用 linux network namespace 隔离，所以在实际中相当于每个 container 使用的单独一个物理网卡。而如果你运行了 Docker，但是用 ip netns list 命令是查看不到有 Docker 的 network namespace 的，但是实际上 Docker 是创建了设备，只是用 ip 命令单纯查看了 /var/run/netns，实际并没有查询内核。

## 3. 架构

Docker 在设计的时候考虑多个场景。典型的使用场景包括使用 image、制作 image、启动多个 container。

### (1) 制作 image

为了制作自己的 image，Docker 提供了 dockerfile 文件格式。Docker 认为要制作一个 image，在大部分情况下需要基于已有的 image。所以在 dockerfile 中定义如下所示内容。

```
FROM docker/whalesay:latest
RUN apt-get -y update && apt-get install -y fortunes
CMD /usr/games/fortune -a | cowsay
```



然后使用 Docker 提供的编译命令, `docker build` 就可以生成自己私有的 image。这个生成的过程本质是启动一个临时的 container, 然后在这个 container 中执行 RUN 后面定义的命令。将生成的 image 重新打包为满足要求的新的 image。如果使用 `COPY ./my-binary /my-binary` 语法就可以把外部的待发布的二进制打包进 images 了。

也可以使用 `docker tag` 命令给一个已经有的 image 进行重新命名, image 的名字包括用户名、image 名字和 tag 这三部分。使用 `docker tag` 命令之后就可以将生成的带 tag 的 image push 到 hub 上了。

## (2) 使用 image

当完成一个 image 的制作, 使用 `docker push` 命令将这个 image 发布到 hub, 这个 hub 就是使用 image 的关键。当你在机器上使用 `docker run`、`docker pull` 等命令运行一个 image 时, 如果本地没有, Docker 会去 hub 服务器上拉取这个 image。也就是说 hub 是 image 在云端存储的地方。

Docker 自带的这种云端管理的能力使得每个公司都可以自己架设自己的私有云, 从而管理企业的 image 资产, 类似一个企业内部的包发布系统。登录这个 hub 需要申请用户名密码以进行验证, 所以 image 的集群管理问题用这种集中式的解决方案解决了。

另外一个维度的使用是实际运行 container。Docker 提供了 `docker logs`, 它可以用来持续地查看我们运行的进程的标准输出。`docker ps` 可以查看当前运行的 container, `docker run/start/stop` 命令可以用来启动和停止一个 container; `docker rm` 命令可以用来删除一个 container; 还可以用 `docker inspect` 命令查看 container 内部的状态。使用这些 Docker 命令就可以完成 Docker 的使用。

## 4. Docker 的安全问题

Docker 官方统计了已经出现的安全漏洞, 定义了 4 个主要的安全维度。

- 内核本身的漏洞 (主要是 namespace 和 cgroup);
- 在平台系统运行的 docker daemon 的漏洞;
- Docker 的配置问题;
- 内核的安全特性配置在 Docker 上的作用 (例如内核 capabilities、TOMOYO、AppArmor、SELinux、GRSEC 等安全机制)。

Docker 系统本身在理论上是安全的, 但是必定存在逃逸漏洞。最重要的安全防护措施就是要做到即使出现了逃逸也不造成损失。典型的措施就是启动 Docker

container 时使用较低的权限。现在的 Docker 还可以将 Docker 内部的 root 映射到外部的非 root 用户，从而也是一个比较明显的特权隔离。对于大部分漏洞而言，Docker 的补救措施都是禁止或者去掉权限，例如 bpf、keyctl、ptrace 这几个高级的函数调用都被直接禁止。因此目前的 docker 仍然称不上安全，因为针对已有的 CVE，很多是采用了禁止和去权限的方法，更有直接指明了不被修复的，例如 CVE-2015-3290, 5157, CVE-2016-5195:。

# 13

## 第 13 章

# 硬件专用于子系统

### 13.1 无线子系统

Linux 内核中有一个 rfkill 子系统，使用这个子系统可以关闭任何一个射频收发器。Linux 中倾向于通用架构子系统，各个设备的作用其实都是实现这个子系统规定的函数。这些子系统向上层就提供操作同类函数的完整接口，这就类似于面向对象编程的 Interface 概念。

无线子系统包括很多内容，例如 WiMAX、红外等，但一般的 Linux 用户都不会用到这些内容。

#### 1. Wi-Fi

Wi-Fi 的特殊之处在于它的应用非常广泛，但竟然是不开源的。就如同显卡一样，硬件被芯片厂商牢牢掌控，对外只提供接口。路由器生厂商拿到的都只是已经封装了 Wi-Fi 固件的接口而已。Linux 中的 Wi-Fi 驱动也是有驱动泄漏或者是逆向获得的通用驱动。一般的 Wi-Fi 芯片都有自己加强版的功能特性，使用内核的驱动就显得力不从心。也就是说当我们选择了开源的驱动时，就得忍受它的低效率。

#### 2. PHY 层

802.11-2007 是目前通用的 2009 的基础版本，之前的过时版本不考虑。802.11-



2009 是较新的版本,目前最普及的 802.11n (100Mb/s) 802.11-2012 就是逐渐开始火起来的 802.11ac,在 5G 的网络中工作,速度很快,但穿透力一般,所以现在的路由器一般会使用双频产品。

802.11-2007 里给出了 5 种 PHY 层,也就是 5 种编码与调制方法,每种 PHY 层对应的 PHY 帧格式都是不同的。也就是说,虽然这个 Wi-Fi 标准对外的接口 (MAC) 是一样的,但是根据底层采用的不同 PHY 层,所以底层从帧格式到编码、调制都是不一样的。5 种 PHY 分别是直序扩频 (DSSS)、跳频扩频 (FHSS)、正交频分复用 (OFDM)、高速率直序扩频 (HR/DSSS)、红外 (IR)。另外,还给出一个叫 ERP 的增强版本的 PHY 层,改变了调制方法,增强了 DSSS 和 OFDM 的速率。

802.11-2009 里又多了一种 PHY 层——高吞吐 (HT OFDM),就是在原来 OFDM 的基础上改进得来的。最新的 802.11ac 里又多了一种 PHY 层——超高吞吐 (VHT OFDM),这个 PHY 层也是通过改进 OFDM 而得来的。它们都支持 MIMO,即多天线,这能提高不少速率。

FHSS 和 DSSS 属于扩频通信,就是把原来在较小带宽传送的信号用较大的带宽来传送。为什么要这样浪费带宽呢?窄带信号容易被干扰,把窄带信号分布到大的带宽上就不易被干扰了。同样的带宽,传输的数据速度不一定谁快。

FHSS 就是在一系列窄频上同步跳跃;DSSS 就是将相对大功率的窄频信号扩展到相对低功率的宽带信号上。两种扩频方式各有优劣,FHSS 在移动性上更好,DSSS 在静止速度上更快。所以在速度为主要考虑因素时,会选用 DSSS;在移动性为主要考虑因素时,会选用 FHSS。目前,Wi-Fi 用户最大的需求是速度,所以选择 DSSS 发展较好。

HR/DSSS 是增强版的 DSSS,原理很简单,只是改变了编码方式和缩减了帧头部长度。从而增加了速度。ERP 通过对 DSSS、OFDM 编码的改进来提高速度。

OFDM 是一种很“神奇”的技术。能提高频谱利用率,简单地说就是调制和复用的结合,提高信道吞吐。而 802.11-2009 里说的 HT OFDM 就是增加了一些 MIMO (多天线) 相关技术。用多根天线来增加速率;802.11ac 里说的 VHT OFDM 就是更高的带宽,更多 OFDM 子载波。在技术上没有本质的变化,都是 OFDM+MIMO 的不断增强。

PHY 层的上层是 MAC,各个版本的 PHY 层需要向 MAC 提供统一的调用接口,就是原语。原语包括以下两类。

(1) 基本特性。包括 MIB 管理 (PLME-GET、PLME-SET)、复位 (PLME-RESET)、

特性参数查询 (PLME-CHARACTERISTICS)、DSSS 进入测试模式 (PLME-DSSSTESTMODE)、发送时间估计 (PLME-TXTIME)。

(2) 数据首发。包括数据传输 (PHY-DATA)、发送控制 (PHY-TXSTART、PHY-TXEND)、信道空闲检测 (PHY-CCARESET、PHY-CCA)、接收控制 (PHY-RXSTART、PHY-RXEND)。

PHY 层的内部也是分层的。最下层叫 PMD (物理介质依赖), 与实际的物理介质打交道; 中间层叫 PLCP (物理层聚合), 它把 MAC 层传下来要发送的数据变成对应的实际物理层 PHY 要发送的数据 (经过调制和编码, 肯定与 MAC 层原来的数据不同了) 发送给 PMD; 最上层叫 PHY SAP, 就是定义的对 MAC 的服务接口。是 MAC 可以直接调用的稳定接口, 不论具体的 PHY 是什么, 这些接口都是可用的, 而且只可以调用这些接口。SAP 就是通过 PLCP 将各个不同的物理介质 PMD 以统一的接口对外展现的。结构中还包括一个信息库 MIB, 负责存储属性参数。

### 3. MAC 层

MAC 层是 802.11 的主要功能部分。上层应用通过调用 MAC 层提供的接口原语调用 MAC 层的功能。

MAC 层一共向上层提供了两大类接口原语 (共 30 种), 如表 13-1 所示 (并非所有的原语都是可调用的, 一部分是 indication 形式的向上通知, 有 request 的是可以调用的)。数据含 1 种, 管理含 29 种。数据部分就是提供普通数据包的收发接口; 管理部分是主要功能部分, 例如发起认证、连接、信道扫描等其他所有管理功能。

表 13-1

数据部分	
数据	MA-UNITDATA
管理部分	
电源管理	MLME-POWERMGT
信道扫描	MLME-SCAN
时间同步	MLME-JOIN
认证	MLME-AUTHENTICATE
断开认证	MLME-DEAUTHENTICATE
建立连接	MLME-ASSOCIATE
重新连接	MLME-REASSOCIATE
断开连接	MLME-DISASSOCIATE

续表

管理部分	
复位	MLME-RESET
网络开始	MLME-START
测量	MLME-MREQUEST
信道测量	MLME-MEASURE
测量报告	MLME-MREPORT
信道切换	MLME-CHANNELSWITCH
发送功率通知	MLME-TPCADAPT
设置密钥	MLME-SETKEYS
删除密钥	MLME-DELETEKEYS
迈克尔失败事件	MLME-MICHAELMICFAILURE
可扩展局域网认证协议帧	MLME-EAPOL
点对点连接请求	MLME-PeerKeySTART
设置发送或接收的安全保护	MLME-SETPROTECTION
帧密钥错误丢弃通知	MLME-PROTECTEDFRAMEDROPPED
交通流 (TS) 管理接口	MLME-ADDTs MLME-DELTs
直接连接管理	MLME-DLS MLME-DLSTearDown
高层同步支持	MLME-HL-SYNC
合并 ACK 帧管理	MLME-ADDBA MLME-DELBA
Qos 调度变更通知	MLME-SCHEDULE
发行商特有	MLME-VSPECIFIC
MIB 管理	MLME-SET MLME-GET

以上所有的原语构成了 MAC 对外提供的可操作接口。

在内部, MAC 层除了有函数还有数据, 叫 MIB, 用于存储 MAC 层的各种参数。它还有个专业术语叫 SME, 其实它是一个单独的模块, 用来跟接口函数进行功能互动, 完成各个函数之间的关联操作和配合响应。属于配合接口正常运作的角色, 对外不提供接口。

以上的接口原语按照功能模块, 可以归纳出 MAC 层主要包括的功能。

(1) 信道管理。包括信道扫描 (MLME-SCAN)、信道测量 (MLME-MREQUEST、



MLME-MEASURE、MLME-MREPORT)、信道切换 (MLME-CHANNELSWITCH)。

(2) 连接管理。包括认证 (MLME-AUTHENTICATE)、断开认证 (MLME-DEAUTHENTICATE)、建立连接 (MLME-ASSOCIATE)、重新连接 (MLME-REASSOCIATE)、断开连接 (MLME-DEASSOCIATE)、开始网络 (MLME-START)、点对点连接请求 (MLME-PeerKeySTART)、直接连接管理 (MLME-DLS、MLME-DLSTearDown)。

(3) 服务质量 (Qos)。包括交通流 (TS) 管理接口 (MLME-ADDTS、MLME-DELTS)、Qos 调度变更通知 (MLME-SCHEDULE)。

(4) 功率控制。包括电源管理 (MLME-POWERMGT)、发送功率通知 (MLME-TPCADAPT)。

(5) 安全。包括密钥管理 (MLME-SETKEYS、MLME-DELETEKEYS)、迈克尔失败事件 (MLME-MICHAELMICFAILURE)、EAPOL (MLME-EAPOL)、帧密钥错误丢弃通知 (MLME-PROTECTEDFRAMEDROPPED)。

(6) 时间同步。包括时间同步 (MLME-JOIN)、高层同步支持 (MLME-HL-SYNC)。

(7) 特性。包括合并 ACK 帧管理 (MLME-ADDBA、MLME-DELBA)、发行商特有 (MLME-VSPECIFIC)、MIB 管理 (MLME-SET、MLME-GET)。

#### 4. 信道接入方式

信道接入看起来是 PHY 层的活儿，但是这是一个算法，不是一个操作，所以是 MAC 层的活儿，信道接入的几种方式就属于 MAC 层的功能。

Wi-Fi 的信道接入模式包括两种，即 CSMA/CA 和节点协调模式。在无 Qos 的情况下，使用两种模式都可以；在有 Qos 的情况下，在两种模式的基础上又分别定义了优先级来实现 Qos 的传输。

CSMA/CA 就是载波监听冲突检测，是最基本的无线接入方式。IEEE 的无线标准大多数都是以这种接入方式为基本方式。就是监听信道，没有其他实例正在传输数据就传，有就随机退避一段时间再监听信道。标准“给”了它一个名字——DCF (Distributed Coordination Function)，意思是协调是分布的，大家平等来商量谁接入网络。

节点协调模式就是无线路由器安排特定的节点在特定的时间通信，从而造成无阻塞的信道。这种方式采用的帧发送间隔比 DCF 小，从而保证在 DCF 和本模式同

时存在的网络中，本模式具有较高的接入优先级。标准“给”它的另外一个名字——PCF (Poing Coordination Function)，意思是由某一个节点来“协调”谁接入网络，这个节点就是 AP。

在有 Qos 的情况下，DCF 变成了 EDCA，PCF 变成了 HCCA。其实意思就是为了实现 Qos 的帧差别对待，给定义了优先级。

## 5. Linux 下 Wi-Fi 编程

写 Linux 用户空间程序时，现在 Linux 官方推荐的唯一编程方式就是基于 Netlink 的 nl80211.h 编程。Netlink 是一种在 Linux 下的用户空间和内核空间通信的方式，传输的都是一个个的帧。用户空间程序通过生成预定义好的结构帧，与内核达到传递消息的目的。

nl80211.h 是一个头文件，也是用户空间调用内核 Wi-Fi 相关功能的接口。其中定义了所有暴露给用户空间的 API 函数索引（不是函数本身），以及这些函数采用的参数的格式和定义。用户程序通过 Netlink 机制，将这些 API 函数索引和对应的参数封装到 Netlink 的帧中，发送给内核。内核解析 Netlink 帧后，读取帧中的内容，就知道用户需要调用哪个函数，以及该函数的参数，从而完成内核功能的调用。

nl80211.h 可以用 libnl-genl 封装接口。这是 libnl 的一个高层扩展，就是说 libnl 也可以直接完成这个库的功能。由于 libnl 的帧种类越来越多，所以就有必要给这些帧进行分类，所以就出现了多个 protocol family。由于 iw (Linux 下的 Wi-Fi 配置程序) 用的这个库，所以想要了解 nl80211.h 的调用方法，推荐看一下 iw 的源代码。

配置 Wi-Fi 系统的大体流程如下（只是逻辑关系，从 iw 源代码中抽取）：

```
nl_socket_alloc(); //生成Netlink的 socket (Netlink相关内容参考上文的介绍)
nl_socket_set_buffer_size(state->nl_sock, 8192,
8192); //调整缓存大小
genl_connect(state->nl_sock) //socket 和内核连接
genl_ctrl_resolve(state->nl_sock, "nl80211"); //genl 的概念，向内核查询
一下协议族的标志
msg = nlmsg_alloc(); //生成要发送往内核的帧（还没有填充内容）
cb = nl_cb_alloc(iw_debug ? NL_CB_DEBUG : NL_CB_DEFAULT); //生成回调函
数，回调函数相关，见第一篇 Netlink 的文档
genlmsg_put(msg, 0, 0, state->nl80211_id, 0, cmd->nl_msg_flags, cmd->cmd,
0); //往刚生成的帧中填充头部信息
NLA_PUT_U32(msg, NL80211_ATTR_IFINDEX, devidx); //向刚生成的帧内部添加一
```

个属性值

```
nl_socket_set_cb(state->nl_sock, s_cb);           //设置回调函数
nl_send_auto_complete(state->nl_sock, msg);       //发送刚生成的帧给内核。自此，内核当收到该请求时就会执行在帧中填充的命令索引和参数。比如搜索无线网，帧中就会填充 scan 命令对应的索引和要扫描的信道作为参数

while (err > 0)
    nl_recvmsgs(state->nl_sock, cb); //等待接收内核的反馈
```

## 13.2 音频子系统

### 1. 音频框架

音频设备是非常常用的，但又是最容易在 Linux 下出现问题的设备之一。音频设备和芯片的种类繁多，所以必须提供足够多的驱动，并且内核有足够的驱动与设备匹配的能力。也正是由于多样性，内核必须向上提供一个统一的接口。于是就有了音频框架。

音频框架与其他框架一样，作用都是向下提供要求，要求每个音频驱动都必须要实现结构体和函数，向上提供统一的操作，使得上层软件只需要调用固定的操作函数、识别固定的结构体就可以操作所有的音频设备。

以前 Linux 内核中的音频框架为 OSS (Open SoundSystem)，后来 Gentoo 开发了用户空间的音频架构 ALSA，内核到 2.6 版本又将其移动到内核，从此内核采用了 ALSA 架构，原来的 OSS 架构标记为“废弃”。

ALSA 向上提供了两种级别的抽象，内核开发者都可以调用，一种是 alsa-lib；另一种是 alsa-soc。SOC 是对 alsa-lib 的再封装。用户端的库也叫 alsa-lib，但编译后叫 libasound。Android 系统还简化修改了这个库，形成了 TinyAlsa 库。更高级的音频封装还有 SDL、OpenAL 等。

### 2. 音频接口

音频接口有很多种，我们现在最常见的是 HD Audio，其次是 AC 97。其他常用的音频接口还有 I2S、PCM。

HD Audio 是最新的标准，是在 2004 年提出的，用来取代 AC 97。HD Audio 与 AC 97 声卡在使用上最大的不同是 HD Audio 的音频接口是可以任意插的，任何一个



音频接口，你可以将它配置为麦克风，也可以将它配置为音响。而 AC 97 是固定的，绿色的插孔只能插音箱，粉色的插孔只能插麦克风，插错了是无法播放声音的。HD Audio 还有一个很强大的功能，就是可以多路输出，例如一台电脑同时玩游戏和听音乐，能够通过配置实现让两组声音从不同的物理 HD Audio 的插口输出声音（首先你得有多个插口，很多机器前面板是 AC 97 的，后面板是 HD Audio 的）。

AC 97 开创双芯片模式，将音频的所有数字部分集成为一个芯片，所有模拟部分集成为一个芯片。HD Audio 也是一样，但提高了能力，输出也可以是数字的。HD Audio 的数字部分叫作 Controller，里面包含了一个或多个 Codec，Codec 是专门用来编解码的。因此大部分的驱动都是在这里工作。

### 3. 音频驱动

Linux 把 alsa 驱动分为了 3 个层次，即 Machine、Platform 和 Codec。Machine 驱动负责 Platform 和 Codec 之间的耦合。其中 Codec 部分内核不需要实现，都是厂商提供的，但是 Linux 系统中的设备驱动普遍采取内核模块的形式以源代码提供，所以常用的 Codec 还是在内核中集成的。Linux 对 HD Audio 的支持还不够完善，只有一个 snd-hda-intel 驱动，还经常出现问题。对于 Codec 驱动，HD Audio 的驱动是 snd-hda-codec，AC 97 的驱动是 ac97\_codec。snd 模块是最基本的 alsa 模块，被所有设备使用。snd-pcm-oss 可以模拟 oss pcm 设备最新的数字部分。

### 4. 音频设备

ALSA 框架规定了很多设备的种类，在 include/sound/core.h 中定义。我们常见的设备有两种：MIDI、PCM 音频。还有其他设备，如定时器、时序器、混音器等。

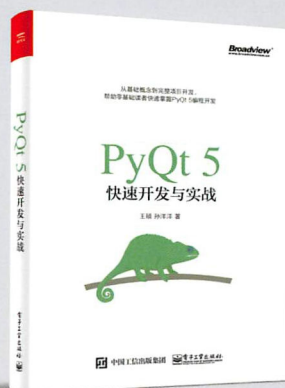
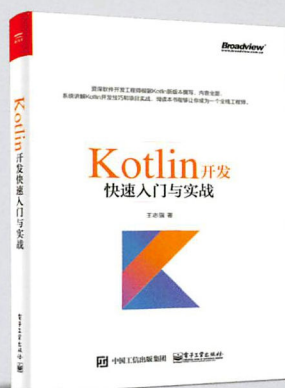
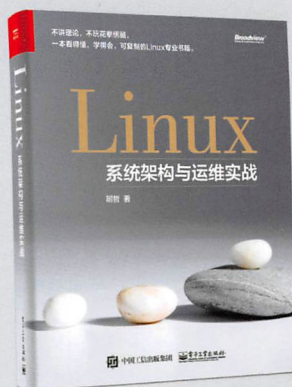
MIDI 是数字化乐器的通信标准，它记录和传输的不是具体的声音波形，而是操作乐器的指令，通过这个指令 MIDI 乐器就可以播放指定的音乐。在 MIDI 乐器上弹奏的音乐也可以以 MIDI 指令的形式存储为文件由内核识别。这种设备家用的较少，一般为专业音乐人员使用。

PCM 音频是家用的音响系统。PCM 本身是一种脉冲调制方式，一般指模拟的音频信号。音乐记录在 PC 上可以是模拟的（PCM 文件），也可以是数字的（MP3 等），但是在播放时，都需要转换为模拟的，然后输出到播放模拟音频信号的设备上播放。对应的录音设备也是一样的原理。

现代计算机系统，一般存储在 PC 上的音频都是数字格式的，是根据采样定理

对模拟信号以高频率采样后获得的与模拟信号可完全互相转化的数字格式的音频，然后再对这个完整的数字格式音频进行转化。我们经常听到无损音频，例如 `ape`、`flac` 格式的文件，还有有损的音频，如 `MP3`、`mpeg` 等。这两种音频都是针对这个数字格式而言的（由于未压缩的数字格式与模拟格式可以互相转化，所以也相当于针对模拟音频而言），采用不同的压缩算法，有的根据语音的特点欺骗人耳，将人耳不能听到或者不敏感的声音区域删除，人耳无法分辨，但已经不是之前的音乐了。

还有一个比较重要的设备接口是 `Controller`，`Controller` 接口主要是让用户空间的应用程序（`alsa-lib`）可以访问和控制音频 `Codec` 芯片中的多路开关、滑动控件等。对于 `Mixer`（混音）来说，`Control` 接口显得尤为重要，从 `ALSA 0.9.x` 版本开始，所有的 `Mixer` 工作都是通过 `control` 接口的 `API` 来实现的。当然，这部分是 `HD Audio` 的东西，`AC 97` 在 `ALSA` 中有兼容的实现。`Controller` 可以控制声音的回放、捕捉、加强音量等。



出版编辑联络：黄爱萍  
 微信 / QQ：69476637  
 邮箱：huangaip@phei.com.cn



在众多介绍Linux内核架构的书籍中，本书的出现犹如Linux内核从多年的2.x跃迁到3.x一样让人兴奋。作者刘京洋江湖人称“刘叔”，是一位学贯古今中外的计算机大咖，他凭借深厚的技术功底将Linux内核架构的原理娓娓道来，使不同层级的读者如读小说一般轻松掌握Linux内核的精髓。

· 苏玉鑫 博士 ·

听闻“刘叔”要出书，我还在猜想主题是科幻小说、商业分析、国际政治、历史解读，还是技术类？印象中，从大学时代，“刘叔”就是这样超人般的存在。这本书又让我想起在嵌入式组时光，大家吃着盒饭，指点江山的样子。

· 郑德权 ·

你我素未谋面，只因“刘叔”这本书在茫茫人海中结缘。关于Linux的书籍，目前以使用型和操作型居多，而思索底层内核原理者甚少。这本书深入浅出地将Linux内核架构与底层原理和盘托出，独树一帜，很多实验和结论都是“刘叔”查阅大量资料反复推敲而来，何不细细品味一番？

· 何聪辉 博士 ·

京洋同学从学生时代开始就是Linux系统的狂热爱好者，经过多年深入研究和思考，对Linux内核形成了深刻、独到的理解。本书浓缩了Linux内核方方面面的知识，同时也融入了作者的点滴经验和感悟，读之犹如与一技术高手在对话，颇受启发。

· 陈剑武 博士 ·

用“十年磨一剑”描述此书再贴切不过。记得十年前，与京洋初次接触Linux时折腾一中午才安装了一个叫Everest的Linux发行版，现在看来当初真的只是管中窥豹。但至此，一发不可收拾，京洋坚持了下来，以自己的见解，逐步积累才得此良作，何不开卷一读？

· 林奕 ·

这是一本令人着迷的书，建议每一个嵌入式开发者都仔细阅读。此书既可以引导读者了解Linux内核的原理，又适合作为一本参考书籍备在手边，相信无论是哪一种阅读方式，都会让你受益匪浅。

· 孙梦石 ·



博文视点Broadview



@博文视点Broadview



策划编辑：黄爱萍  
责任编辑：徐津平  
封面设计：侯士卿

上架建议：操作系统 > Linux

ISBN 978-7-121-32290-7



定价：89.00元